KFA

KERNFORSCHUNGSANLAGE JÜLICH GmbH

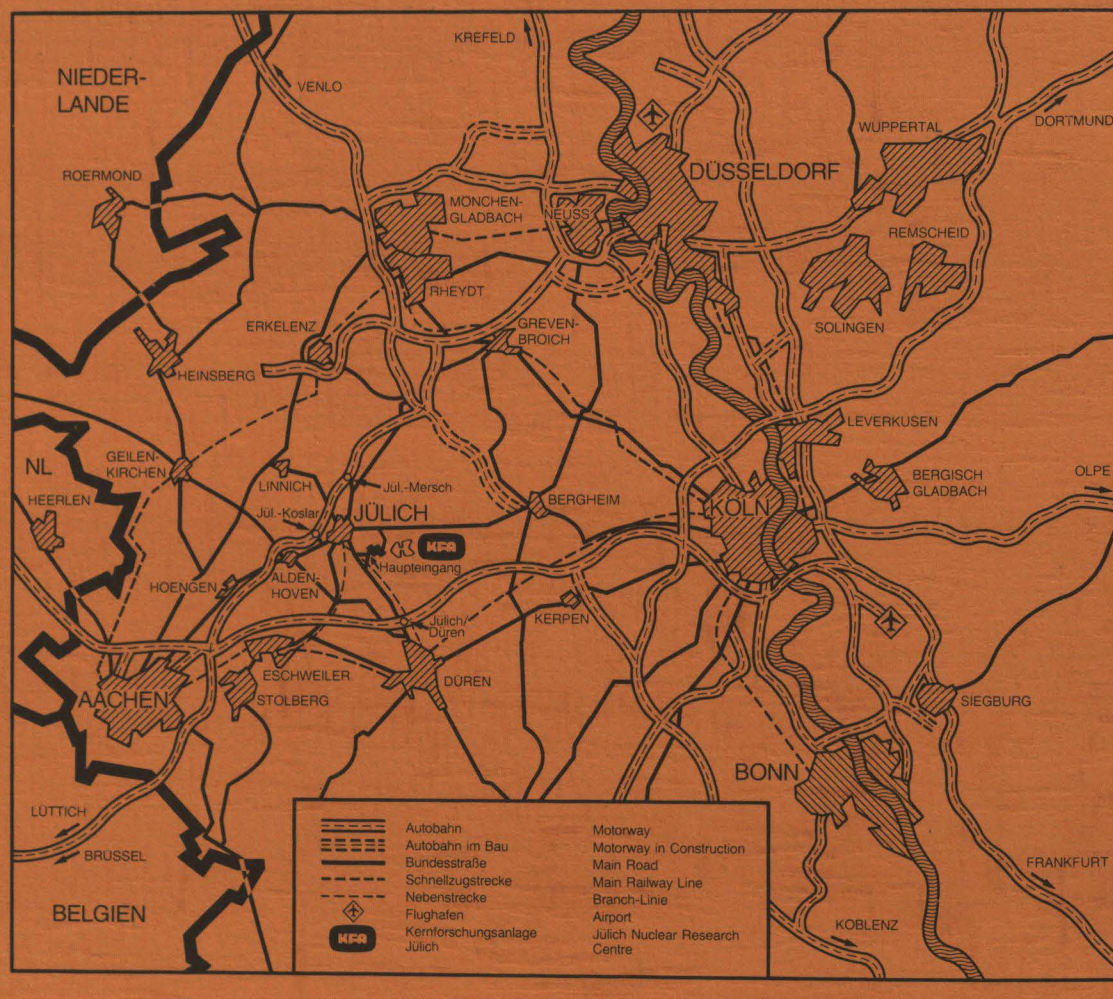
Zentralinstitut für Angewandte Mathematik

Pipeline-Rechner: Architektur und Programmierung

von

H.-D. Winckens

Jül - Spez - 164
Juli 1982
ISSN 0343 - 7639



Als Manuskript gedruckt

Berichte der Kernforschungsanlage Jülich – Nr. 164

Zentralinstitut für Angewandte Mathematik Jül – Spez – 164

Zu beziehen durch: ZENTRALBIBLIOTHEK der Kernforschungsanlage Jülich GmbH
Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)
Telefon: 02461/610 · Telex: 833556 kfa d

Pipeline-Rechner: Architektur und Programmierung

von

H.-D. Winckens

Diese Arbeit wurde als Diplomarbeit am Institut für Allgemeine Elektrotechnik und Datenverarbeitungssysteme bei Prof. Dr.-Ing. O.Lange im Zentralinstitut für Angewandte Mathematik angefertigt.

Inhaltsverzeichnis

1. Einleitung

Bereits seit Beginn des Computerzeitalters und ungeachtet der eindrucksvollen Entwicklung in der modernen Rechnergestaltung hat immer ein Bedarf an leistungsfähigeren Rechnern als den gerade verfügbaren bestanden. Dieser Wunsch stützt sich bis heute nicht zuletzt auf die von Anwendern an die Hersteller herangetragenen Forderungen nach Lösungsmöglichkeiten für immer komplexer werdende Probleme, deren Lösung auf Rechnern konventioneller Art nur mit großen Schwierigkeiten, häufig aber gar nicht durchzuführen sind.

Beispiele solch neuer Problem-Dimensionen lassen sich aus vielen Bereichen der Wissenschaft und auch der Wirtschaft angeben :

- Wettervorhersage
- Radarüberwachung
- Festkörperphysik
- Bildverarbeitung
- Computer Aided Design (CAD)
- Operations Research

Als konventionell sind jene Strukturen zu bezeichnen, die auf der sequentiellen von-Neumann Architektur basieren, und die beispielsweise die IBM 360/370 Serien sowie viele Computerentwürfe anderer Hersteller noch weitgehend bestimmen.

Um eine Steigerung der Leistungsfähigkeit zu erreichen, wurden in der Vergangenheit hauptsächlich Fortschritte im Bereich der Technologie unter Beibehaltung der konventionellen Struktur ausgenutzt. Unter Leistungssteigerung soll hier in erster Linie die Erhöhung der Bearbeitungsgeschwindigkeit von Instruktionen und Operationen verstanden werden, die für das sogenannte "number crunching", das heißt, die Verarbeitung großer Mengen von Gleitpunktoperationen, auf wissenschaftlichem Sektor besonders wichtig ist.

Der Gedanke an neuartige Strukturkonzepte mit mehr Hardware-Aufwand ist nicht neu, gewann aber mit der radikalen Verbesserung des Preis-Leistungs-Verhältnisses auf technologischem Sektor immer mehr an Bedeutung.

Kapitel 2 gibt nach einer kurzen Übersicht über den heute erreichten Stand der Technologie einen Überblick über die architektonischen Konzepte, die aus der Forderung nach immer leistungsfähigeren Rechnern resultieren.

Eines dieser Konzepte, das Prinzip des Pipelining, wurde bereits in den 60-er Jahren eingesetzt, um Rechnern eine schnellere Bearbeitung der gestellten Aufgaben zu ermöglichen und gilt für moderne effiziente Rechnersysteme als entscheidende Architekturform. Kapitel 3 beschäftigt sich eingehend mit diesem Pipeline-Konzept auf dem Gebiet der Rechnerarchitektur und stellt einige neuere Pipelinesysteme vor.

In Kapitel 4 werden Problematiken bei der Programmierung von Pipelinerechnern angesprochen und Lösungsmöglichkeiten für spezielle Probleme auf dem FPS AP190L - Pipelinerechner untersucht. Dies umfaßt in erster Linie rekursive Probleme, deren Aufbau eigentlich dem Prinzip des Pipelining entgegenläuft.

Der Anhang beinhaltet zwei Assemblerprogramme zur Lösung spezieller rekursiver Probleme auf dem AP190L.

2. Architektur digitaler Rechnersysteme

2.1 Definition: Rechnerarchitektur

Die Diskussion neuartiger Rechnerkonzepte macht es erforderlich, daß zunächst der Begriff "Architektur" mit seiner Bedeutung für den weiteren Verlauf der Untersuchungen festzulegen ist. Das hierarchische Schichtenmodell eines Rechners (Bild 2.1) bietet verschiedene Möglichkeiten für die Definition der Architektur aus den unterschiedlichen Betrachtungsebenen heraus.

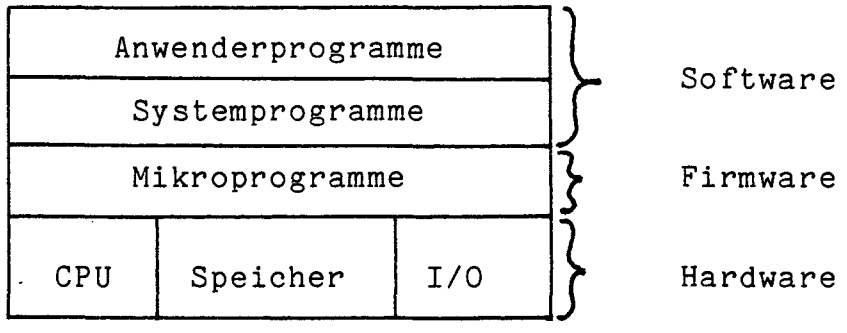


Bild 2.1: Schichtenmodell eines Rechners

Die Schnittstellen zwischen den einzelnen Schichten bilden diese Betrachtungsebenen. Wie unterschiedlich der Begriff "Architektur" aufgefaßt werden kann, zeigen einige Zusammenstellungen solcher Definitionen /ReF 76/, /BoH 80/. Blaauw /BoH 80/ befindet sich bei seiner Definition auf einer sehr hohen Ebene, indem er die Architektur als funktionales Erscheinungsbild des Rechners dem Benutzer gegenüber deklariert.

Dieser Benutzer sieht das System beim Entwurf und bei der Bearbeitung seiner Anwenderprogramme jedoch nur durch die "Brille der höheren Programmiersprachen" und besitzt daher von der eigentlichen Arbeitsweise des vorliegenden Rechners nur bedingt Kenntnis.

Im Gegensatz dazu existieren aber auch sehr tief - in Bezug auf das Schichtenmodell - angesiedelte Definitionen. So versteht zum Beispiel Stone /BoH 80/ die Architektur als Zusammenschluß der einzelnen Hardwarekomponenten wie Prozessoren, Speicher, E/A-Geräte und Verbindungseinheiten.

Diese verschiedenartigen Definitionen des Begriffs "Architektur" lassen sich ,ähnlich wie bei Myers /Mye 78/, in einem System von Betrachtungsebenen zusammenfassen und sind zu vergleichen mit den Schnittstellen zwischen den Schichten des Modells aus Bild 2.1. Auf der obersten Ebene stellt sich das gesamte System aus der Sicht des Benutzers dar, d.h., hier werden die einzelnen Funktionen des Systems als grobstes Architekturkonzept des Rechners zusammengefaßt, ohne der eigentlichen Realisierung dieser Funktion Beachtung zu schenken.

Diese Ebene bildet die Schnittstelle zwischen Benutzer und Gesamtsystem und trägt die Bezeichnung "Systemarchitektur".

Die nächst niedrige Betrachtungsebene ermöglicht den Einblick in die Handhabung logischer (z.B. Files, virtuelle Speicher) und physikalischer (z.B. Haupt- und Sekundärspeicher) Betriebsmittel. Dies macht auch der Begriff "Softwarearchitektur" deutlich, der diese Schnittstelle zwischen Anwenderprogrammen und System- programmen kennzeichnet.

Einer weiteren Schnittstelle kommt hier eine besondere Bedeutung zu, und zwar der Ebene zwischen der Software auf der einen sowie der Firm- und Hardware auf der anderen Seite. Sie beinhaltet die Darstellung des Systems aus dem Blickwinkel der Software.

Hier liegt der eigentliche Bereich der "Computerarchitektur", der charakterisiert werden kann als Definition eines physikalischen Systems "aus der Sicht eines Programmierers, der seine Programme in einer Maschinensprache verfaßt" /Mye 78/ und so unmittelbaren Zugang zu den einzelnen Hardwarekomponenten erhält. Wenn im Folgenden von der Architektur eines Rechners gesprochen wird, so soll diese Form der Computerarchitektur darunter verstanden werden.

Es besteht natürlich die Möglichkeit,auch tiefer liegende Betrachtungsebenen als Definitionsgrundlagen heranzuziehen, so daß immer detailiertere Definitionen der Hardwarebausteine und deren Verschaltung auftreten. Diese Aspekte sollen hier jedoch nicht weiter verfolgt werden, da sie für die weitere Behandlung des Themas nicht relevant sind.

2.2 Die von-Neumann Architektur

Das Konzept der grundlegenden Architekturform, allgemein bekannt als klassischer von-Neumann-Rechner, wurde in den Jahren 1946/47 zum ersten mal veröffentlicht. Dies geschah durch die theoretischen Arbeiten von Burks, Goldstein und von Neumann, die damals an der Universität in Princeton/USA tätig waren. Jahrzehnte beherrschte dieses Konzept entscheidend den Computermarkt, wohl aufgrund seiner vielfältigen Einsatzmöglichkeiten und der trotzdem eingehaltenen Unkompliziertheit.

Eine einfache Form der Darstellung dieses Fundamentalrechners liefert die Unterteilung in 5 Teilbereiche (Bild 2.2) :

- Das Leitwerk (LW) steuert den Programmablauf und kontrolliert den Fluß von Daten und Instruktionen.

- Im Rechenwerk (RW) werden arithmetische Operationen und logische Verknüpfungen durchgeführt.

- Der Speicher (SP) dient der Aufnahme von Programm und Daten.

- Das E/A-Werk stellt die Verbindung zwischen dem eigentlichen Rechner und den peripheren Geräten (Außenwelt) her.

- Die Verbindungseinheit (BUS) ermöglicht die Kommunikation der einzelnen Teilbereiche untereinander.

Die Arbeitsweise dieser Maschine läßt sich durch 2-stufige Arbeitsgänge beschreiben:

i) Der Speicherzelleninhalt der im Befehlszähler (Teil des LW) angegebenen Adresse wird in das Leitwerk geholt und als Befehl interpretiert.
ii) Die in diesem Befehl enthaltenen Adressen bestimmen das Holen weiterer Speicherzelleninhalte, wobei diese jetzt als Daten von dem Typ angenommen werden, der im Befehl gefordert wird.

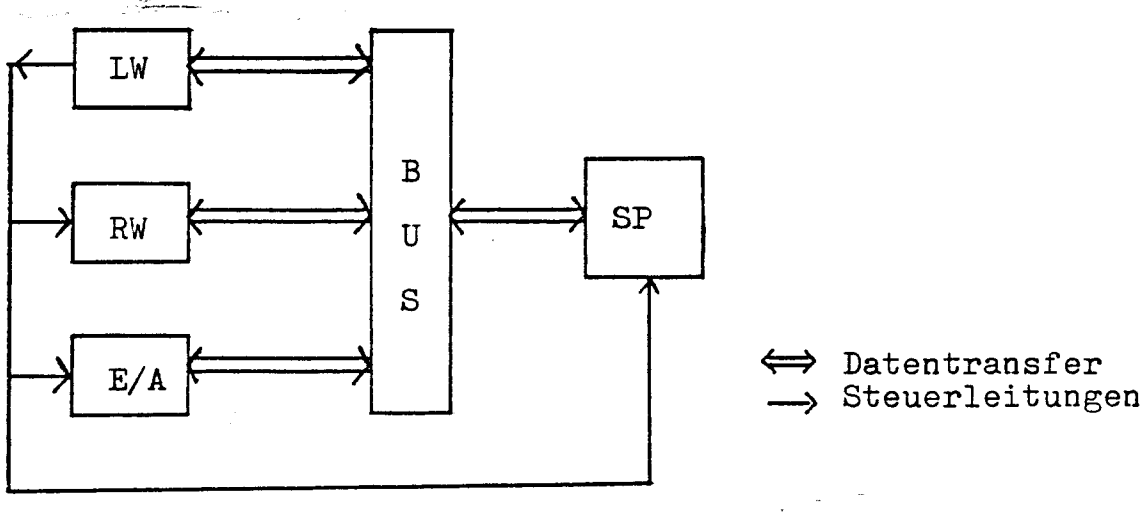


Bild 2.2: Blockbild des von-Neumann-Rechners

Unter Berücksichtigung dieser Arbeitsweise lassen sich einige nachteilige Eigenschaften dieses Konzeptes erkennen:

- Programm und Daten werden in einem einzigen eindimensionalen Speicher mit sequentieller Adressierung abgelegt.
- Die Maschine selbst hat keine Möglichkeit zu entscheiden, ob ein vorliegendes Bitmuster eines Speicherwortes ein Datum oder einen Befehl repräsentiert.
- Ebenso ist es ihr nicht ersichtlich, welche Bedeutung die einzelnen Daten im Speicher haben. So läßt sich nicht feststellen, ob z.B. eine Gleitkommazahl oder eine Zeichenkette vorliegt.

Zusammenfassend sind diese Punkte als "Prinzip des minimalen Speicheraufwandes" und das gesamte von-Neumann-Konzept als "Architektur des minimalen Hardware-Aufwandes" /Gil 81/ zu beschreiben.

Ein weiteres gravierendes Merkmal der von-Neumann-Maschine besteht in dem sogenannten "von-Neumann-Flaschenhals" /Bac 78/. Hiermit ist der Engpaß beim Datentransfer zwischen CPU und Speicher gemeint. In erster Linie ist das Ziel eines Programmes die Datenveränderung im Speicher, und daher müssen immer wieder einzelne Daten durch diesen Flaschenhals gepumpt werden. Ein großer Teil dieses Verkehrs besteht jedoch nicht aus "nützlichen Daten", sondern aus Namen bzw. Adressen von Daten, die den Engpaß blockieren und eine Steigerung der Bearbeitungsgeschwindigkeit hemmen.

Dieses Problem resultiert nicht zuletzt aus der strikt sequentiellen Abarbeitung von Rechnerinstruktionen, die über die reine Hardware-Architektur hinausgehend auch die Struktur der Programmiersprachen bestimmt. Besonders bei der Behandlung größerer Datenmengen kommt dieser Nachteil in entscheidendem Maße zum Tragen.

Aus all den aufgeführten Eigenschaften des fundamentalen von-Neumann-Konzeptes läßt sich die Forderung nach neuartigen Rechnern ableiten, um neue Problemdimensionen in schnellerer und somit akzeptabler Zeit bearbeiten zu können.

Wie läßt sich aber ein Rechner "schneller machen" ? Als Antwort sind drei unterschiedliche Vorgehensweisen anzuführen:

- Verbesserung der Technologie
- Änderungen in der Architektur
- effektivere Algorithmen

Kombinationen aller drei Möglichkeiten versprechen natürlich den größt möglichen Erfolg. Das Hauptaugenmerk sei hier auf architektonische Möglichkeiten gelegt. Zuvor jedoch soll ein kurzer Exkurs einen Überblick über den heutigen Stand der Technologie geben, um bezüglich Leistungsfähigkeit und Kostenentwicklung der Hardware-Bausteine eine Basis für die Realisierung neuer Systeme zu erhalten.

2.3 Stand der Technologie

Die zwei Hauptbereiche der Halbleitertechnologie, die zur Implementierung grundlegender logischer Schaltkreise in Rechnern dienen, sind:

- bipolare Halbleiter
- Metalloxid-Halbleiter (MOS-FET Technik)

Bei den bipolaren Halbleitern treten in der Basis-Emitter-Grenzschicht Ladungsträger beider Polaritäten, positive Löcher und negative Elektronen, auf. Dagegen verwendet die sogenannte MOS-Technik (Metall-Oxid-Silizium) spannungsgesteuerte Feldeffekttransistoren (FET). Sie zeichnet sich durch hohe Integrationsdichte und geringen Leistungsbebarf aus, hat allerdings wesentlich niedrigere Schaltgeschwindigkeiten als die bipolaren Techniken /Kla 75/.

Einige charakteristische Daten dieser beiden weit verbreiteten Techniken sind in Tabelle 2.1 zusammengestellt. Weitere starke Verbesserungen verspricht man sich von dem sogenannten Josephson-Effekt und von der GaAs-Technologie (Gallium-Arsenid), die sich aber noch in der Entwicklungsphase befinden.

Im Vergleich der Werte aus Tabelle 2.1 mit den Daten aus vergangenen Jahren läßt sich erkennen, daß die Entwicklungen in den letzten 20 Jahren die Integrationsdichten um den Faktor 1000 (Bild 2.3) und die Verarbeitungszeiten um den Faktor 100 verbessern konnten /Hoß 80/, /Gil 81/.

Betrachtet man außerdem die Preisentwicklung, so ergibt sich eine Verbesserung des Preis-Leistungsverhältnisses um nicht weniger als den Faktor 16.000 /Hoß 80/. Da die hier genannten Tendenzen auch in der nächsten Zukunft noch anhalten werden, wenn auch gewisse Grenzen schon abzusehen sind, ist das Prinzip des minimalen Hardware-Aufwandes, wie es der von-Neumann-Rechner verkörpert, nicht mehr allein gültig, und die Entwicklung neuer Architekturen gewinnt wesentlich an Bedeutung.

Tabelle 2.1: Vergleich Bipolar - MOS

	Bipolartechnik	MOS-FET
Logik-bausteine	Geschw.: 1-10 ns Dichte : einige 100 c/c	Geschw.: 10-100 ns Dichte : > 1000 c/c
Speicher-elemente	Zugriff : 20-200 ns	Zugriff : 60-300 ns

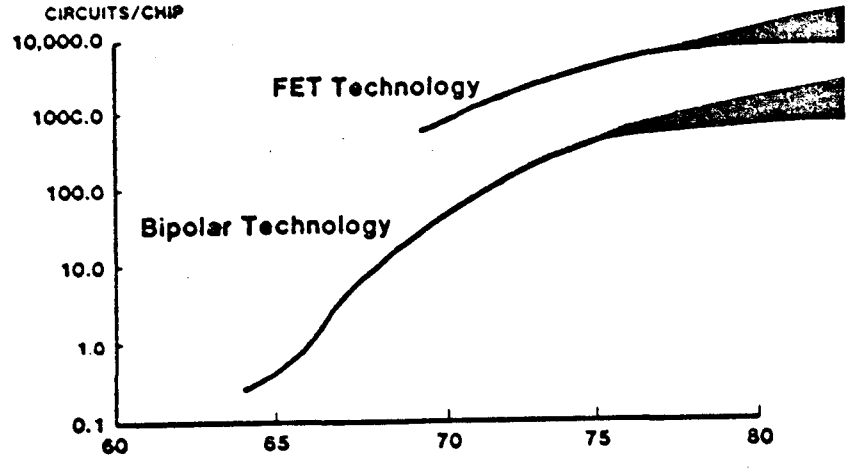


Bild 2.3: Entwicklung der Integrationsdichte bei Logik-Schaltungen

2.4 Innovative Architekturformen

2.4.1 Einführung

Die Entwicklung neuartiger Architekturkonzepte verfolgt das Ziel, die Lösung von Problemen schneller und/oder effizienter durchführbar zu machen, als es auf Rechnern konventioneller Bauart möglich ist. Dies soll jetzt, bei gleicher Technologie, durch mehr Hardwareaufwand erreicht werden. Um solche Leistungsverbesserungen feststellen zu können, dienen die beiden Vergleichsmaße "Speedup" und "Effizienz" /Kuc 78/.

Definition: Speedup

A sei eine bestimmte Berechnungsvorschrift und $T_1(A)$ die Zeit, die ein konventioneller, streng sequentieller Monoprozessor zur Ausführung von A benötigt. Entsprechend sei die Zeit, die sich zur Berechnung von A für einen Rechner mit neuartigem Architekturkonzept ergibt, gleich $T_N(A)$ mit N gleich der Anzahl der eingesetzten Hardwareelemente (z.B.: Anzahl der parallelen Prozessoren oder Anzahl der Pipelinestufen). Dann ist der Speedup (= Steigerung der Berechnungsgeschwindigkeit) bezüglich A definiert durch

$$S_N(A) := \frac{T_1(A)}{T_N(A)} \qquad (2.1)$$

und es folgt, für $S_N(A) > 1$ liegt eine echte Geschwindigkeitserhöhung vor.

Definition: Effizienz

Die Effizienz eines Computersystems für eine Berechnungsvorschrift A ergibt sich als Quotient aus dem Speedup $S_N(A)$ und N

$$E_N(A) := \frac{S_N(A)}{N} \qquad (2.2)$$

2.4.2 Feldrechner (Array-Processor)

Der Grundgedanke der Array-Organisation ist die Zusammenschaltung einer beliebigen Anzahl von identischen Prozessorelementen (PE) unter der Leitung einer einzigen zentralen Kontrolleinheit. Die einzelnen Prozessorelemente sind untereinander verbunden, - zum Beispiel mit ihren nächsten Nachbarn (Bild 2.4) - besitzen jeweils einen lokalen Speicher und arbeiten synchron, das bedeutet gleichzeitige Ausführung der gleichen Operation auf unterschiedlichen Daten.

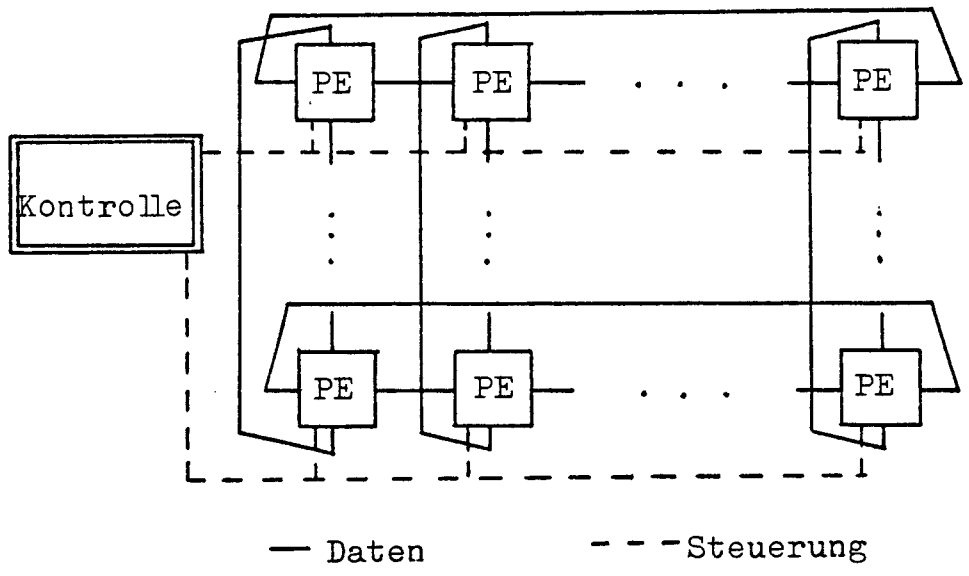


Bild 2.4: Beispiel einer geschlossenen Arraystruktur

Ein solcher Arrayrechner könnte Anwendung finden im Bereich der Bildverarbeitung und Mustererkennung sowie bei vielen numerischen Problemen mit matrixförmiger Strukturierung der zu verarbeitenden Daten. Daher bilden die Arrayrechner eine Teilmenge der "special-purpose-computer" und sind meistens durch die Kontrolleinheit an einen "general-purpose-computer", dem sogenannten Host-Rechner, angeschlossen.

Zur gleichen Zeit kann also ein Arrayrechner maximal N Operationen ausführen, wenn N die Anzahl der zusammengeschalteten Prozessorelemente angibt, und es ergibt sich ein maximal erreichbarer Speedup von

$$S_N(A) = N \qquad (2.3)$$

Dies ist jedoch nur in den wenigsten Fällen zu erzielen, da die Struktur der zu verarbeitenden Daten dann exakt der Form des Prozessorfeldes entsprechen muß und zusätzlich keine Konflikte bei der Bereitstellung der Daten auftreten dürfen.

Einer der ersten und wohl einer der am häufigsten angeführten Vertreter dieser Architekturform ist der ILLIAC IV - Rechner, der Ende der 60-er Jahre entwickelt wurde /BBK 68/.

Die ILLIAC IV ist ein Arrayrechner mit 4 quadratischen Matrizen zu je 8x8 Prozessorelementen, wobei jeder dieser Quadranten eine eigene Kontrolleinheit besitzt. Trotz der ausgezeichneten Kritiken in der Literatur, die die ILLIAC IV als "Meilenstein in der Computerentwicklung aufgrund ihres hohen Parallelitätsgrades" /Thu 76/ bezeichnen, wurde nur ein Prototyp mit nur einem Quadranten realisiert.

Weitere Beispiele für Rechnersysteme vom vorgestellten Array-Typ sind:

SOLOMON (Westinghouse)
Massive Parallel Processor MPP (Goodyear Aerospace)
Distributed Array Processor DAP (ICL)

2.4.3 Multiprozessorsysteme

Multiprozessorsysteme zeichnen sich aus durch eine Vielzahl kompletter CPUs, die sowohl über private wie auch über einen zentralen Speicher verfügen können. Man unterscheidet Systeme mit zentraler und dezentraler bzw. verteilter Kontrolle.

Liegt eine zentrale Kontrolle vor, so übernimmt ein ausgezeichneter Prozessor die gesamte Systemaufsicht und kontrolliert die Arbeit der ihm untergeordneten Prozessoren - "Master-Slave-Prinzip". Hier sind die Prozessoren ausschließlich stark miteinander gekoppelt (tightly coupled), das heißt, sie haben Zugriff auf einen gemeinsamen Speicher (memory sharing). Im Gegensatz dazu ist bei Systemen mit verteilter Kontrolle auch eine schwache Kopplung möglich (loosely coupled). Dies bedeutet, die Kommunikation der Prozessoren beschränkt sich auf den Austausch von Botschaften über eine gemeinsame Kommunikationseinrichtung (message switching). Bei diesem Prinzip der verteilten Kontrolle, auch "kooperative Autonomie" genannt, sind die Betriebssystemfunktionen von jedem Prozessor wahrnehmbar, und die Systemsteuerung wechselt zwischen diesen.

Das erstgenannte Prinzip ist in der Regel effizienter, das zweite aber stabiler gegenüber den Störungen, die sich durch Hard- und Softwarefehler ergeben können.

Als wichtige Charakteristika für Multiprozessorsysteme sind anzugeben:

- keine fest vorgegebene Struktur
- mehrere Prozesse können nebeneinander ablaufen

Aus der variablen Struktur läßt sich auch der Begriff "Parallelprozessor-Ensemble" erklären, der solche Rechner treffend beschreibt. Zwei Beispiele für Realisierungen solcher Systeme sind die symmetrischen Multiprozessorsysteme C.mmp /WaB 72/ und Cm* /SFS 77/, die an der Carnegie Mellon University/USA entwickelt wurden. Als symmetrisch bezeichnet man jene Systeme, deren Prozessoren vergleichbar sind bezüglich ihrer Aufgabe im Gesamtsystem.

Eines der neuesten Systeme mit zentraler Kontrolle ist das HEP-Multiprozessorsystem der amerikanischen Computerfirma Denelcor aus Denver/Colorado /Smi 79/.

2.4.4 Pipelinerechner

Der Grundgedanke der Pipeline-Architektur ist das Prinzip der Fließbandverarbeitung So wird eine komplexe Funktion oder Operation F in eine Reihe einfacherer und kürzerer Teilfunktionen bzw. Teiloperationen zerlegt (Segmentierung), so daß jede einzelne Teilberechnung durch ein spezielles Bearbeitungselement ausgeführt wird (Bild 2.5) und eine kürzere Abarbeitungszeit als die gesamte Funktion erfordert.

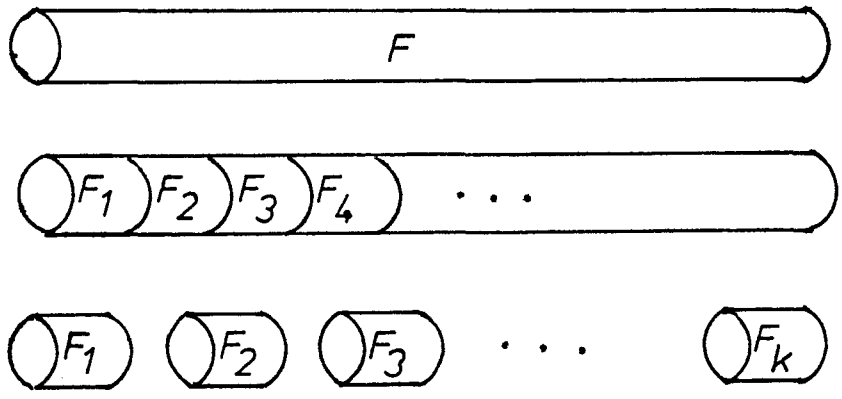


Bild 2.5: Segmentierung

Wird die $F_1,...F_k$ realisierende Folge von Bearbeitungselementen mit einer festen Taktperiode versehen, so kann bei jedem Takt das Zwischenergebnis aus einem Bearbeitungselement in das nächste geschoben werden, während am Anfang dieser Pipeline neue Operanden in das System eintreten. Ist die Pipeline gefüllt, also alle Bearbeitungselemente beschäftigt, so erhält man nach jedem Takt ein Ergebnis am Ausgang des F_k ausführenden Elementes.

Zahlreiche Operationen in Rechenanlagen können auf die oben gezeigte Weise segmentiert werden. Als Beispiel sei hier aus den arithmetischen Operationen die Gleitpunktaddition F genannt. Eine mögliche Segmentierung spaltet diese Operation in 4 Teiloperationen auf:

F_1: Exponent vergleichen
F_2: Mantissen verschieben
F_3: Mantissen addieren
F_4: Ergebnisse normalisieren

Die Arbeitsweise einer solchen Pipeline läßt sich sehr anschaulich darstellen durch ein Raum-Zeit Diagramm für die einzelnen Bearbeitungsstufen (Bild 2.6).

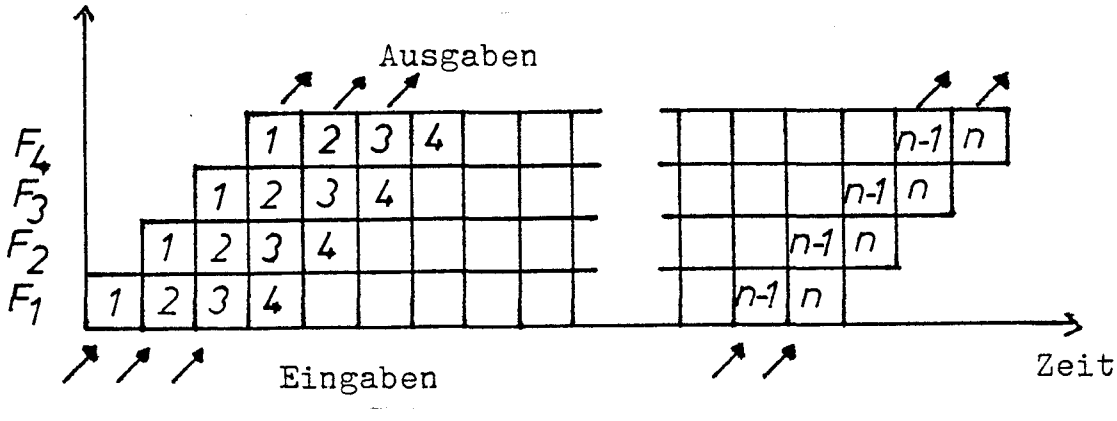


Bild 2.6: Raum-Zeit Diagramm für eine 4-stufige Pipeline und n-fache Ausführung von F

Unter der Annahme, daß, abgesehen vom größeren Hardwareaufwand, die Zeit T_k für die Ausführung einer Operation durch eine Pipeline mit k Segmenten der Zeit T_1 entspricht, die für eine Funktionseinheit ohne Segmentierung erforderlich wäre, so ergibt sich für die n-malige Ausführung der entsprechenden Operation ein Speedup von

$$S = \frac{T}{T_k} = \frac{nk}{k+n-1} \qquad (2.4)$$

und es gilt: $\lim_{n \to \infty} S = k$.

Reale Systeme mit diesem Pipelinekonzept wurden von einer Reihe von Herstellern entwickelt, so unter anderen die Systeme CRAY-1, CYBER 205, IBM 3838 und FPS AP 190 L, die im Kapitel 3 noch näher vorgestellt werden.

2.5 Klassifizierung unterschiedlicher Architekturen

Die Entwicklung immer neuer Rechnersysteme mit unterschiedlichen Architekturformen und Arbeitsweisen fürte zu Versuchen, diese Vielzahl von Systemen nach Merkmalen zu ordnen und in bestimmten Schemata zusammenzufassen. Drei Beispiele hierfür werden im Folgenden kurz erläutert.

2.5.1 Klassifizierung nach Flynn

Einer der Ersten, der sich mit dem Problem der Rechnerklassifizierung eingehend beschäftigte, war Michael J. Flynn /Fly 72/. Seine Klassifizierung ist am weitesten verbreitet, wohl aufgrund ihres relativ einfachen und klaren Aufbaus. Flynn beschreibt eine Maschinenstruktur aus makroskopischer Sicht und unterscheidet 4 grobe Klassen von Organisationsformen nach der Möglichkeit der gleichzeitig zu verarbeitenden Instruktionen und Daten:

a) Single Instruction - Single Data (SISD)

   Diese Klasse verkörpert alle konventionellen Rechner des von-Neumann Typ, da hier nur einzelne Instruktionen auf einzelnen Daten bzw. Datenpaaren ausgeführt werden können.

b) Single Instruction - Multiple Data (SIMD)

   Unter diese Kategorie fallen die Arrayrechner mit einer Kontrolleinheit und entsprechendem Array von Prozessorelementen, die zur gleichen Zeit alle die gleiche Instruktion ausführen müssen. Ebenso sind einzelne Pipelines SIMD-Organisationen, da zu einem Zeitpunkt mehrere Daten durch eine Instruktion bearbeitet werden. Dies geschieht zwar durch mehrere

separate Teilbearbeitungen, jedoch steht nur eine einzige Instruktion zur Verfügung.

c) Multiple Instruction - Multiple Data (MIMD)

Neben allen Multiprozessorsystemen sind auch komplexere Array-Strukturen (mit mehreren Kontrolleinheiten) wie auch die heutigen Pipelinerechner dieser MIMD-Klasse zuzuordnen, da jeweils unterschiedliche Operationen auf unterschiedlichen Daten gleichzeitig durchführbar sind.

d) Multiple Instruction - Multiple Data (MISD)

Diese Klasse muß als leer angesehen werden, da zur Berechnung eines einzigen Datenwertes bereits ein ganzer Befehlsstrom notwendig wäre.

Ein entscheidender Nachteil dieser Flynn'schen Klassifizierung wird bereits durch diese kurze Beschreibung deutlich: für alle verschiedenen neuen Strukturen, die über die von-Neumann Architektur hinausgehen, stehen nur zwei Klassen zur Verfügung, während eine weitere praktisch leer ist. Dies liefert kein befriedigendes Ergebnis bezüglich der Einordnung neuer Konzepte und führte dazu, daß andere Kriterien zur Klassifizierung herangezogen wurden.

2.5.2 Klassifizierung nach Händler

Händler /Hae 77/ versucht, durch die Angabe bestimmter Kenngrößen der einzelnen Systeme ein Klassifizierungsschema zu finden, was aber eigentlich nichts anderes als eine Möglichkeit der formalen Systembeschreibung ist.

Drei grundlegende Größen charakterisieren demnach ein Rechnersystem:

k: Anzahl der unabhängig arbeitenden Programme im System

d: Anzahl der arithmetisch/logischen Einheiten (ALU) pro Kontrolleinheit

w: Anzahl der Bitpositionen, die parallel durch eine ALU bearbeitet werden

Weitere drei Größen geben Auskunft über die im System vorhandenen Möglichkeiten des Pipelining:

k': Anzahl der Einheiten, die getrennt voneiander verschiedene Teile eines Problems bearbeiten können (Makropipelining)

d': Anzahl der Funktionseinheiten, die gleichzeitig arbeiten können (Instruktionspipelining)

w': Anzahl der Pipelinestufen der arithmetischen Einheit (arithmetisches Pipelining)

Ein Rechnersystem läßt sich dann darstellen als

$$RS = (k*k', d*d', w*w'),$$

und als Beispiel ergibt dies für die ILLIAC IV, die bereits in Kapitel 2.4.2 vorgestellt wurde:

$$RS = (1*1 , 64*1 , 64*1)$$

Das Produkt der sechs Faktoren kann als ungefährer Grad an Parallelität angesehen werden und ermöglicht so auf diesem Gebiet einen Vergleich zwischen verschiedenen Systemen. Allerdings läßt sich nur unter Vorbehalt eine direkte Beziehung zwischen dem Parallelitätsgrad und der Rechenzeit zur Lösung eines Problems angeben, da dies von der Struktur des jeweiligen Problems abhängt.

Schließlich kann das Klassifizierungsschema interpretiert werden als Menge von Punkten im 6-dimensionalen Raum natürlicher Zahlen N , und jedes System wird durch einen diskreten Punkt verkörpert. Diese Interpretation rechtfertigt die anfangs gemachte Aussage, daß eher eine formale Systembeschreibung durch ein 6-Tupel als eine eigentliche Klassifizierung vorliegt.

Wie die Kapitel 2.5.1 und 2.5.2 zeigen, bestehen erhebliche Schwierigkeiten, zufriedenstellende Klassifizierungen anzugeben. Dies beruht darauf, daß bei der Klassifizierung das Kriterium einer guten Merkmalswahl darin besteht, daß die Merkmale helfen müssen, eindeutige Zuordnungen zu disjunkten Klassen zu finden. Dagegen nimmt eine "Taxonomie" in Kauf, daß die erhaltenen Gruppierungen nicht leere Durchschnitte haben können.

2.5.3 Taxonomie nach Kuck

Kuck gibt in seiner Arbeit /Kuc 80/ eine Übersicht über bestehende oder mögliche Architekturkonzepte für Hochgeschwindigkeitsrechner. Er unterscheidet zunächst vier fundamentale architektonische Prinzipien, die eine Geschwindigkeitssteigerung gegenüber konventionellen Rechnern ermöglichen:

a) Multiprozessoren
b) Multifunktions-Prozessoren
c) Parallelprozessoren
d) Pipeline-Prozessoren

ad a): Beim reinen Multiprozessor ist eine Verbindung mehrerer konventioneller Rechner zu einem Gesamtsystem vorgesehen.

ad b): Mit Multifunktions-Prozessoren sollen bessere Verarbeitungszeiten erzielt werden, indem man einige Funktionseinheiten eines konventionellen Rechners (z.B.: Addierer, Multiplizierer, logische Einheiten ...) mehrfach auslegt. Eine erweiterte Kontrolleinheit verteilt dann Aufgaben auf die einzelnen Funktionseinheiten und steuert die gleichzeitige Abarbeitung.

ad c): vgl. Kapitel 2.4.2 Feldrechner

ad d): vgl. Kapitel 2.4.4 Pipelinerechner

Die Realisierungen der letzten zehn Jahre zeigen, daß vor allem Kombinationen aus diesen vier Grundmethoden sehr gute Geschwindigkeitsergebnisse erzielt haben; so lassen sich zum Beispiel segmentierte Funktionseinheiten zu einem Multipipelinerechner zusammenfassen oder ähnliches.

Eine Übersicht über die Grundprinzipien und den sich daraus ergebenden Kombimationen mit Beispielen für realisierte Systeme des entsprechenden Typs vermittelt Bild 2.7. Zusätzlich werden hier die architektonischen Konzepte charakterisiert nach der Art ihrer Datenbearbeitung. Das heißt, es wird unterschieden, ob Skalare oder Felder von Daten verarbeitet werden, und es ergibt sich eine Einteilung in drei Maschinentypen, die wiederum der Flynn'schen Klassifizierung sehr verwandt erscheint:

1. single execution array (SEA) Maschinen

2. multiple execution scalar (MES) Maschinen

3. multiple execution array (MEA) Maschinen

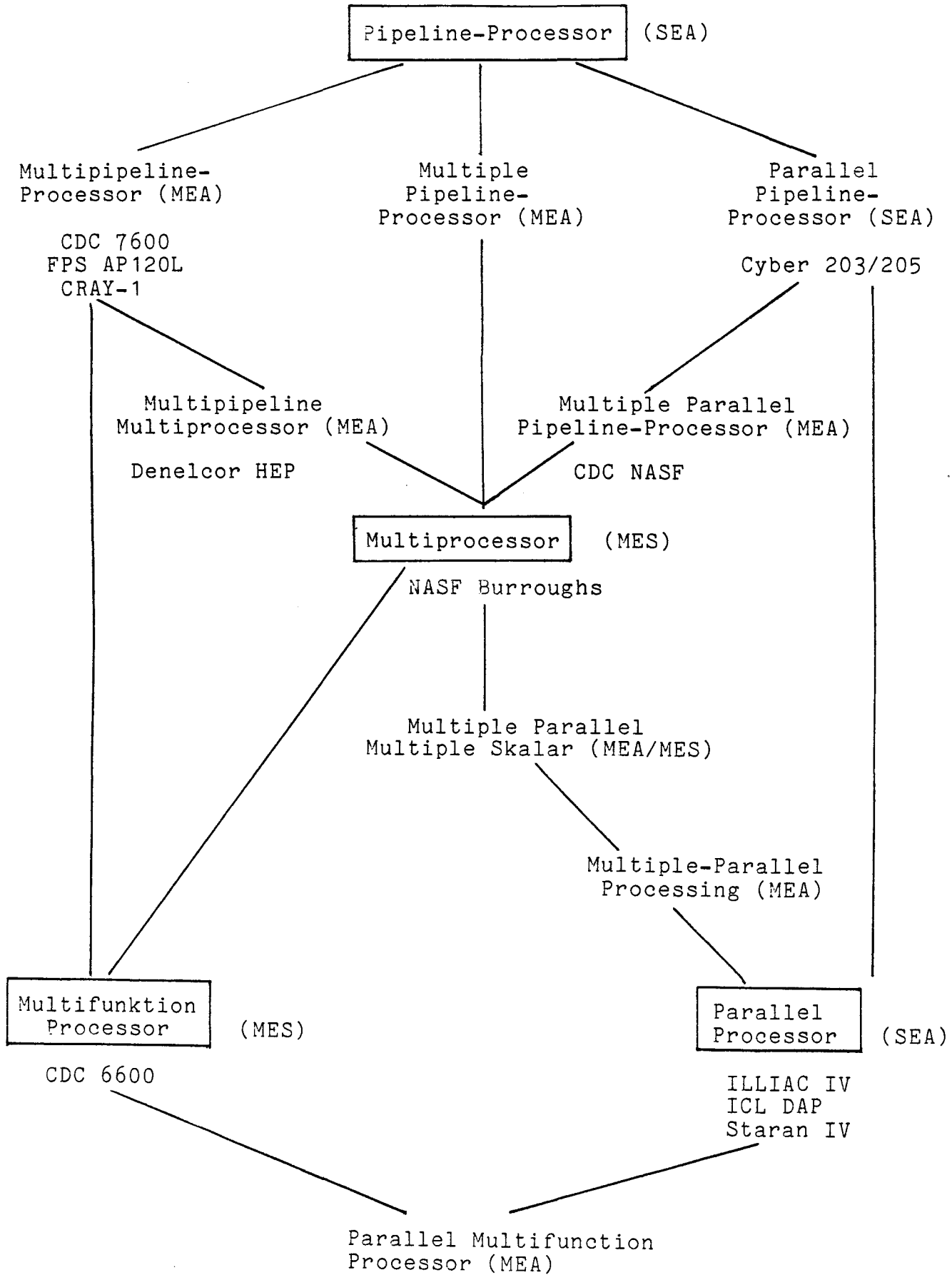


Bild 2.6: Rechnertaxonomie nach Kuck

3. Pipeline-Rechner

3.1 Einführung

Wie bereits im vorangegangenen Abschnitt 2.5.3 deutlich wurde, wird die Entwicklung von Hochgeschwindigkeitsrechnern neben dem Parallelitätsgedanken maßgeblich bestimmt durch die Idee des Pipelining. Nahezu die Hälfte aller von Kuck angegebenen innovativen Architekturen zur Erreichung höherer Rechengeschwindigkeiten (vgl. Bild 2.7) verwenden in irgend einer Form das Pipeline-Konzept.

Der Oberbegriff "Pipelining" beinhaltet drei unterschiedliche Formen:

1. Makropipelining
2. Instruktionspipelining
3. Funktionspipelining

ad 1): Werden gewisse Funktionseinheiten eines Rechners, komplette Rechenwerke oder sogar vollständige Prozessoren zu einer Pipeline zusammengeschaltet, um Operationen auszuführen, die wesentlich komplexer sind als die elementaren Operationen eines Prozessors, so spricht man in diesem Falle von einer "Makropipeline".

ad 2): Beim Instruktionspipelining wird der Befehlsabarbeitungs-Zyklus in - im allgemeinen 4 - verschiedene Phasen unterteilt /Gil 81/:

a) Befehl holen
b) Befehl interpretieren
c) Operanden holen
d) Befehl ausführen

Der Zeitaufwand für die Interpretationsphase ist, zumindest bei einem großen Teil der Operationen, gering gegenüber der Zeit für die Phasen, die den Speicher frequentieren. Da aber, wie in Kapitel 2.4.4 bereits angedeutet, ein ungefähr gleicher Zeitbedarf für alle Pipelinestufen gefordert wird, verspricht eine etwas geänderte Unterteilung eine bessere Annäherung an diese Forderung (Bild 3.1):

a) Befehl holen (BH)
b) Operanden holen (OH)
c) Befehl ausführen (BA)
d) Ergebnisse abspeichern (ES)

Werden diese einzelnen Phasen von unabhängigen Einheiten bearbeitet, so besteht die Möglichkeit der unabhängigen Ausführung von vier Instruktionen, wobei sich jede in einer unterschiedlichen Phase befindet (Bild 3.2).

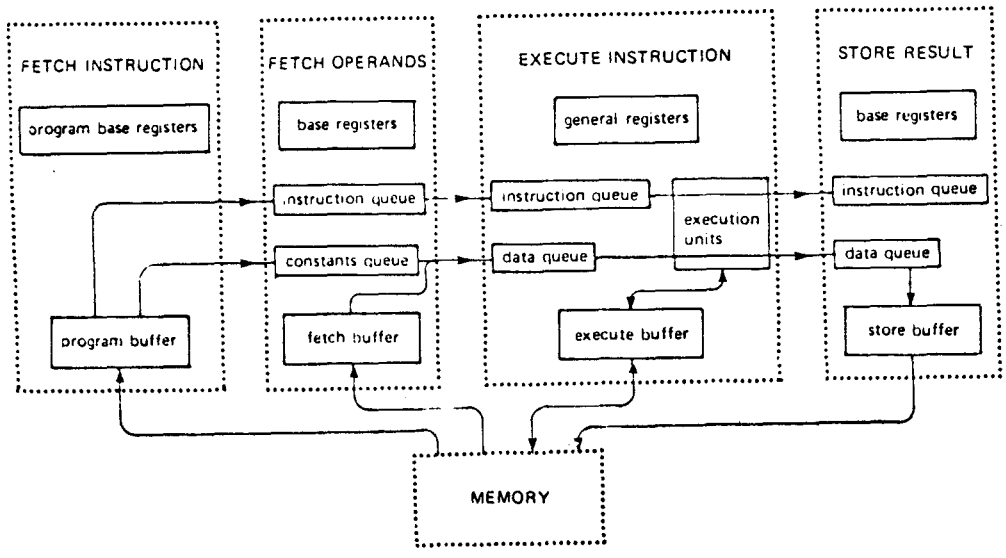


Bild 3.1: idealisierter Instruktionspipeliner /Dor 79/

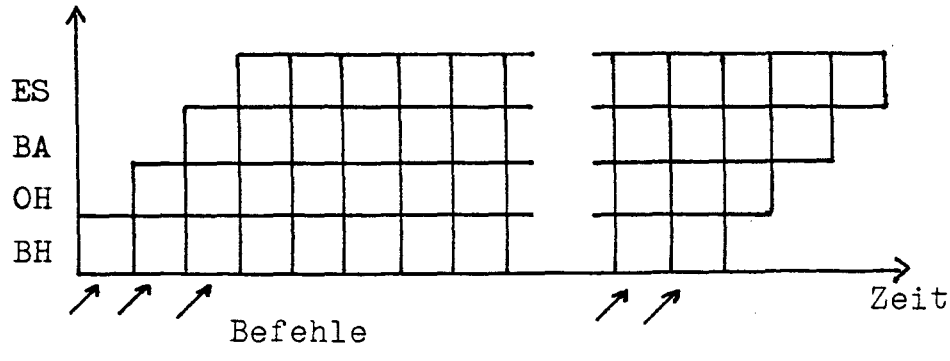


Bild 3.2: Raum-Zeit Diagramm für Instruktionspipelining

Die so mögliche schnelle Abarbeitung der Befehle wird natürlich empfindlich gestört, sobald ein Sprungbefehl auftritt. Denn dann kann erst der Folgebefehl geholt werden, wenn der Sprungbefehl ausgeführt ist. Dies führt zu einer Verzögerung von 2 Phasenlängen (Phasenlänge = Takt).

Angenommen, im Mittel sind 20 Prozent aller Befehle Sprungbefehle /Gil 81/, die jeweils einen Zeitbedarf von zwei Takten zusätzlich erfordern, so beträgt die mittlere Ausführungszeit eines Befehls 1.4 Takte. Das kommt nahezu einer Geschwindigkeitssteigerung um den Faktor 3 gegenüber der sequentiellen Abarbeitung gleich.

ad 3): Das Funktionspipelining wurde bereits in Kapitel 2.4.4 anhand der Segmentierung einer Gleitpunktoperation vorgestellt. Die Vorteile und Probleme, die als Pipeline organisierte Funktionseinheiten mit sich bringen, bilden den Schwerpunkt der folgenden Betrachtungen.

Zunächst jedoch soll ein kurzer historischer Rückblick zeigen, daß die Idee des Pipelining keinesfalls neu ist, sondern bereits in verschiedenen Rechnern älteren Datums als Mittel zur Geschwindigkeitssteigerung herangezogen wurde.

3.2 Historie

Einer der ersten Rechner mit einer gewissen Überlappung von Programmausführung und Ein-/Ausgabeaktivitäten, was als grobes Makropipelining angesehen werden kann, war die UNIVAC 1 (UNIVersal Automatic Computer).

Der Schritt zum Instruktionspipelining wurde gegen Ende der fünfziger Jahre durch die Systeme STRETCH und LARC vorbereitet. Hier wurde zum ersten Mal eine Aufteilung des Instruktionsausführungsprozesses in zwei Stufen vorgenommen: die erste Stufe bestand aus dem Holen des Befehls und seiner Decodierung, die zweite Stufe diente dann der Ausführung des Befehls. Weitere Fortschritte machten sich in der Entwicklung des IBM 360/Model 91 /IBM 67/ bemerkbar. Unter Verwendung einer Pipelinehierarchie wurden weitere Geschwindigkeitsverbesserungen erzielt.
Pipelinehierarchie bedeutet, daß die einzelnen Segmente oder Stufen, aus denen sich die gesamte Instruktionsbearbeitung zusammensetzt, selbst wiederum als Pipeline realisiert sind.

Zu Beginn der siebziger Jahre setzte sich dann auch, nicht zuletzt infolge der neuen technologischen Möglichkeiten, die Segmentierung der einzelnen Funktionseinheiten durch und führte zur dritten Form des Pipelining, dem Funktionspipelining.

Die Systeme STAR 100 von CDC und ASC von Texas Instruments seien hier stellvertretend für diese Rechnergruppe genannt.

Als modernste Pipelinesysteme gelten heute Großrechner wie CRAY-1, CYBER 203/205 und IBM 3838 AP, die als Weiterentwicklungen der vorgenannten Rechner anzusehen sind, und deren erste Exemplare fast gleichzeitig im Jahre 1976 die Produktion verließen.

3.3 Die Hardware

3.3.1 Klassifizierung von Pipelines

Nach der Unterscheidung von drei verschiedenen Formen des Pipelining (Kap. 3.1) sollen jetzt die physikalischen Pipelines selber nach bestimmten Merkmalen ihrer Konfiguration charakterisiert und entsprechenden Klassen zugeordnet werden.

Zwei Arten der Klassifizierung bieten sich an:

1) Klassifizierung nach den Berechnungsmöglichkeiten, die eine Pipeline bietet:

   a) Einfunktionspipeline
   das heißt, die Pipeline bearbeitet für alle möglichen Eingaben immer dieselbe Operation. Ein typisches Beispiel hierfür ist die Pipeline zur Gleitpunktaddition in Kap.2.4.4.

   b) Mehrfunktionspipeline
   das heißt, mehrere unterschiedliche Funktionen lassen sich durch eine einzige Pipeline bearbeiten. Hier muß dann über die Eingabe eine gewisse Kontrolle gesteuert werden, um sicher zu stellen, daß jeweils auch die richtige Funktion oder Operation ausgeführt wird.

2) Klassifizierung nach dem aktuellen Gebrauch einer Pipeline:

   a) statische Pipeline
   das heißt, für längere Datenströme wird die zu bearbeitende Funktion nicht geändert.

   b) dynamische Pipeline
   das heißt, eine Änderung der zu bearbeitenden Funktion tritt häufig auf. Dies ist zum Beispiel beim Instruktionspipelining der Fall, denn es muß jeweils eine andere Instruktion ausgeführt werden.

Verbindungen zwischen diesen beiden Klassifizierungen lassen sich sehr leicht herstellen, denn eine Einfunktionspipeline ist immer statisch konfiguriert, eine Mehrfunktionspipeline hingegen kann statisch oder dynamisch sein.

3.3.2 Elemente und Zeitverhalten einer Pipeline

Bislang wurde über Pipelines ausschließlich als Einheit gesprochen, in diesem Abschnitt soll jedoch der technische Aufbau einer Pipeline im Vordergrund stehen.

Eine Pipeline besteht, wie bereits mehrfach erwähnt, aus mehreren Stufen, die, jeweils autonom, eine Teilfunktion berechnen. Eine solche Stufe besteht im allgemeinen aus zwei Elementen:

- einem Logikteil zur Ausführung der gewünschten Teilfunktion (logic)
- einem Puffer, welcher die Ausgabe des einen Logikteils als Eingabe des nächsten sichert (latch).

So müssen also für eine Taktzeit, die den Datentransfer von einer Stufe zur anderen regelt, die beiden Zeiten T und W berücksichtigt werden (Bild 3.3). T bezeichnet den den Zeitbedarf des Logikteils, W den Zeitbedarf des Latch für Aufnahme und Weitergabe der Resultate und P die Summe von W und T.

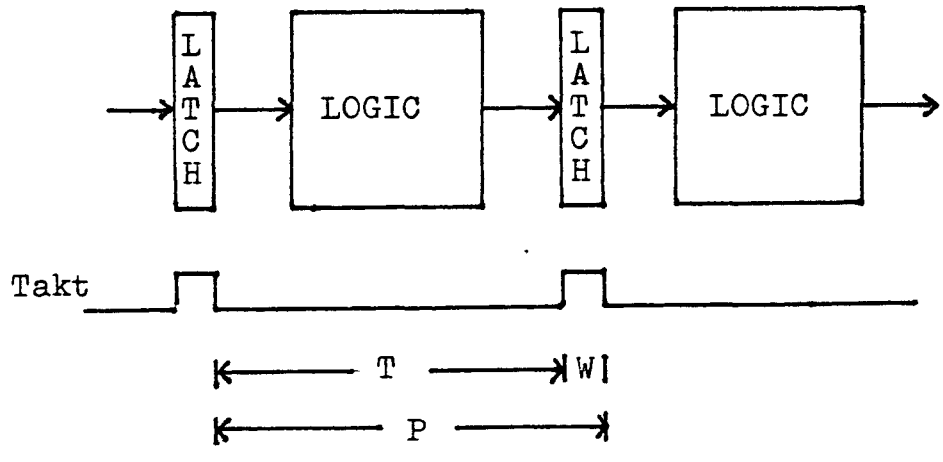


Bild 3.3: Pipelinetakt

An die Zeit t, die letztlich zur Ausführung des Logikteils zur Verfügung steht, sind jedoch noch einige zusätzliche Bedingungen zu knüpfen. Ist $t_{max} \geq t \geq t_{min}$, so gilt trivialerweise $t_{max} \leq T$. Ist der Zeitbedarf der Logik sehr kurz, so kann es passieren, daß das neue Ergebnis noch im gleichen Takt zur nächsten Logik durchgeschaltet und so das Gesamtergebnis verfälscht wird. Daher muß der zu kurze Pfad durch die Logik durch "do-nothing circuits" soweit verlängert werden, daß gilt: $t_{min} \geq W$.

Berücksichtigt man weiterhin, daß der Takt nicht alle Stufen gleichzeitig erreicht, zum Beispiel aufgrund der unterschiedlichen Pfadlänge, so muß ein Faktor S, der diese Zeitverschiebung ausgleicht, in die Berechnung einbezogen werden, und es ergibt sich die Ungleichung

$$W + S \leq t \leq T - S$$

für den effektiven Zeitbedarf des Logikteils einer Pipelinestufe.

Vor allem bei kurzen Logikzeiten spielt der Zeitbedarf pro Latch für die Taktfrequenz und somit für die Bearbeitungsgeschwindigkeit einer Pipeline eine wichtige Rolle. Als klassischer Latch-Schaltkreis gilt ein Flip-Flop (Bild 3.4).

Einige Nachteile eines solchen Latches sind erhöhter Hardwareaufwand zur Bereitstellung der negierten Eingaben und ein Zeitaufwand von immerhin 3 Gatterzeiten bis zum stabilen Ergebnis. 1965 stellte dann J. Earle einen neuen Latch-Schaltkreis vor (Bild 3.5).

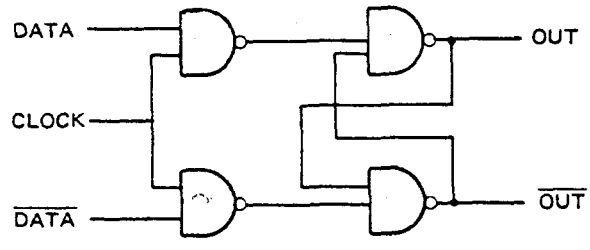


Bild 3.4: Flip-Flop aus NAND-Gattern

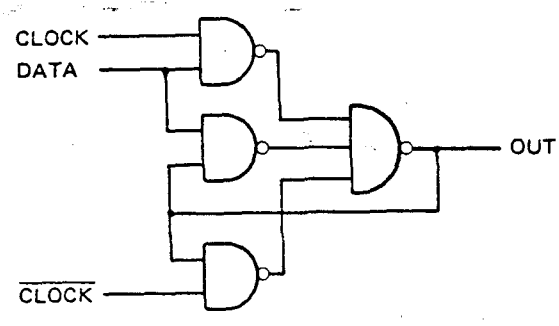


Bild 3.5: Einfache Version des Earle-Latch

Hier besteht nur noch ein Zeitbedarf von 2 Gatterzeiten und nur der Takt muß negiert verfügbar sein.

Es gibt jedoch auch eine Alternative zu dem Gebrauch solcher Zwischenspeicherelemente, welche auf Beobachtungen bei herkömmlichen Pipelineröhren basiert. Demnach ließe sich unter Fortlassen der Latches eine wesentlich höhere Durchsatzrate erzielen. Eine Pipelineröhre, in der Flüssigkeit transportiert wird, benötigt nichts, was einem Latch entspricht, da die Moleküle eines Stoffes diesen Stoff auf dem Weg durch die Pipeline zusammenhalten. Allerdings muß die an den Rändern auftretende Reibung und die sich dadurch ergebende Mischung (Bild 3.6) berücksichtigt werden.

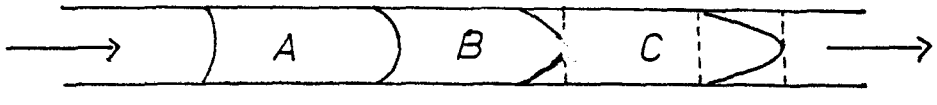


Bild 3.6: Reibungseffekte in einer Pipelineröhre

Ähnlich sieht das Problem dann auch bei elektronischen Pipelines aus. Der Reibung entsprechen die nie exakt gleichen Gatterzeiten und die unterschiedlichen Längen der Verbindungsstrecken. Das heißt aber: es existieren nur bestimmte Zeitabschnitte ,in denen der gewünschte Wert am Ende einer Pipelinestufe vorliegt. Diese Zeitabschnitte werden immer kleiner, je weiter die Daten durch eine Pipeline und somit durch eine Folge von logischen Ebenen geschoben werden.

Bezeichnen $d_{max\ i}$: die maximal benötigte Zeit für Segment i
$d_{min\ i}$: die minimal benötigte Zeit für Segment i
und P die gesamte Taktzeit,
so ergibt sich im k-ten Takt

$$P(k) = P - \sum_{i=1}^{k} (d_{max\ i} - d_{min\ i}) \qquad (3.1)$$

als Zeitraum, in dem die richtigen Teilergebnisse vorliegen. Irgendwann kann also auch $P(k) \leq 0$ auftreten, so daß die angestrebte Lösung eines Problems nicht mehr durchführbar ist. Aus diesem Grunde wird ein Minimum m an Zeit vorgegeben, für dessen Dauer der exakte Wert jeweils vorliegen soll.

Es existiert aber genau ein k, so daß gilt: $P(k) \leq m$. Das heißt, nach k - 1 Takten muß die Berechnung durch einen zusätzlichen Schaltkreis unterbrochen werden, der die Synchronisation übernimmt und die Teilergebnisse wieder für P Sekunden stabilisiert.

Eine nach der beschriebenen Weise realisierte Pipeline bezeichnet man als "Maximum Rate Pipeline", da so eine höchst mögliche Durchsatzrate erzielt werden kann.

Um jedoch solch hohe Geschwindigkeiten auch erreichen zu können, genügt nicht allein die Beschleunigung innerhalb der einzelnen Pipeline, sondern das "Umfeld" muß der hohen Geschwindigkeit natürlich ebenfalls Rechnung tragen. Unter Umfeld sei hier in erster Linie die Datenbereitstellung und die Sicherung der Ergebnisse verstanden. Aus diesem Grunde kommt der Organisation der Speicher eine besondere Bedeutung zu.

3.3.3 Speicherorganisation

Da in den meisten Pipelinerechnern mehrere als Pipeline organisierte Funktionseinheiten parallel arbeiten können, ist eine Rate von bis zu 10 Speicherzugriffen pro Takt durchaus realistisch. Berücksichtigt man außerdem, daß ein solcher Speicherzugriff zum Teil erheblich mehr Zeit in Anspruch nimmt als ein Takt, wird deutlich, daß die konventionelle, rein sequentielle Speicherorganisation bei weitem nicht mehr ausreicht.

Um dieses Problem zu lösen, wird der Hauptspeicher in Speicherbänke fester Größe aufgeteilt. Dies bietet den Vorteil, daß gleichzeitig in jeder Bank auf ein Wort zugegriffen werden kann.

Diese sogenannte Speicherverschränkung kann auf zwei Arten (Bilder 3.7,3.8) durchgeführt werden:

1) einfache Verschränkung
N sequentielle Adressen, i,i+1,i+2,...i+N-1, befinden sich in N verschiedenen Moduln, das heißt, jeder Speichermodul beinhaltet nur Adressen der Form k·N+i für 0≤k≤M-1, wobei M gleich der Anzahl der Speicherworte pro Modul ist.

Sei $M = 2^m$ und $N = 2^k$. Dann können mit Hilfe von m Adressbits 2^k Wörter gleichzeitig aus dem Speicher ausgelesen werden und k weitere Adressbits regeln dann die Reihenfolge der Abarbeitung.
Diese Art der Verschränkung ist jedoch nur zufriedenstellend für typisch sequentielle Datenabarbeitung, wie zum Beispiel bei Vektoroperationen. Wesentlich variabler gestaltet sich der Datenzugriff bei der komplexen Verschränkung.

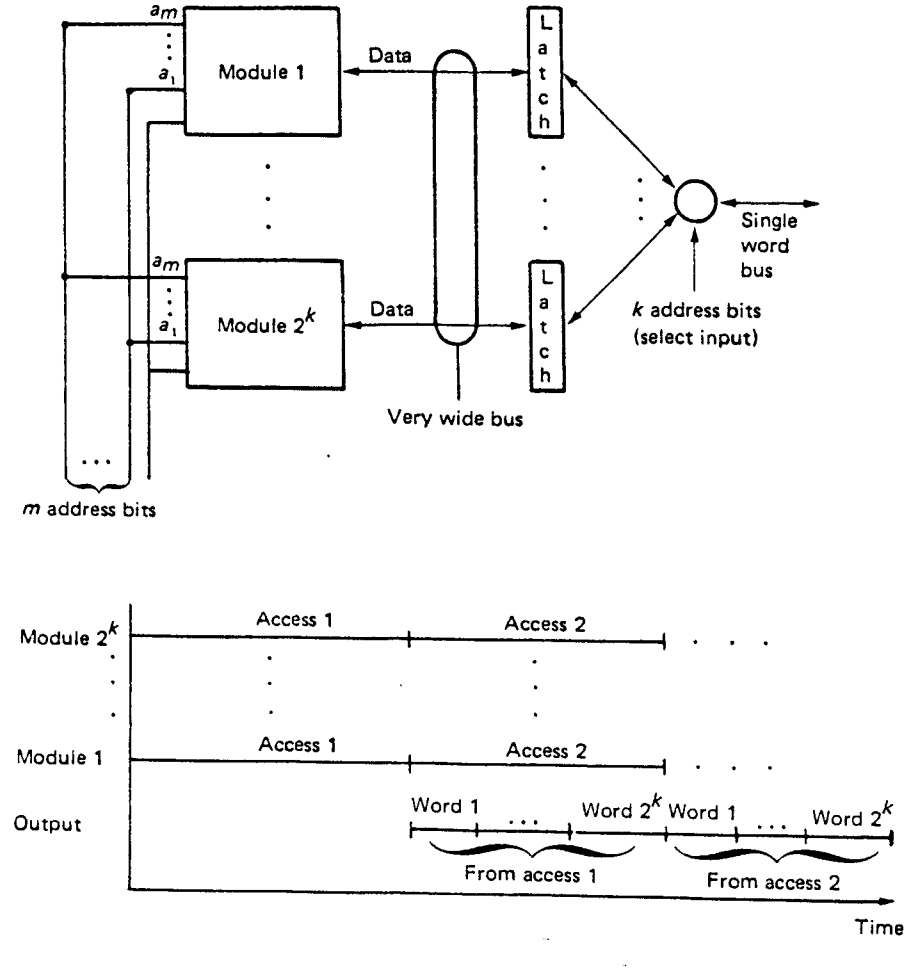


Bild 3.7: einfache Speicherverschränkung

2) komplexe Speicherverschränkung
Die Belegung der Moduln erfolgt wie unter 1). Der Vorteil dieser Art der Verschränkung besteht nun darin, jeden Modul mit einer unterschiedlichen relativen Adresse anzusprechen. Diese Flexibilität wird ermögicht durch eine Speicherkontrolleinheit (Memory Controller). Sie entscheidet, ob der Zugriff auf ein bestimmtes Wort durchgeführt werden kann. Sobald ein Speicherwort von einer Pipeline angefordert wird, prüft sie, ob der entsprechende Modul gerade einen Speicherzugriff bearbeitet oder verfügbar ist.

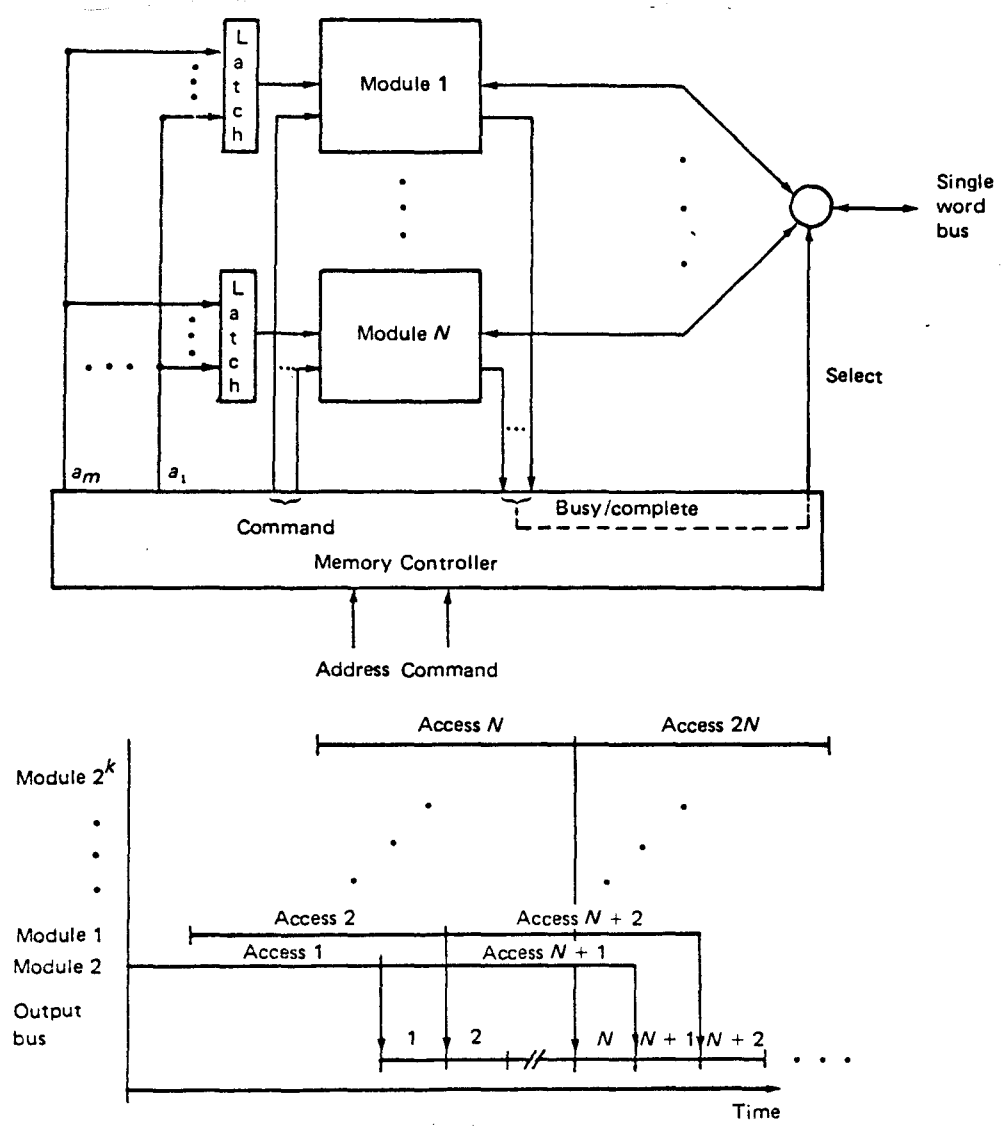


Bild 3.8: komplexe Speicherverschränkung

Dies wird durch eine busy/complete-Meldung an den Memory Controller erreicht. Ist der gewünschte Modul verfügbar, so wird dieser mit der gewünschten Adresse angesprochen und ein Lesezyklus angestoßen (Bild 3.8). Sei N die Anzahl der komplex verschränkten Speichermoduln und T die Zugriffszeit für jeden Modul in Sekunden, so ergibt sich im günstigsten Fall eine maximale Zugriffsrate mit einem Zeitbedarf von T/N Sekunden pro Speicherwort /Kog 81/.

Die Verschränkung des Hauptspeichers allein reicht jedoch noch nicht aus, um den Anforderungen einer Pipeline gerecht zu werden. Mittels einer solchen Verschränkung besteht zwar die Möglichkeit, die Elemente eines Vektors sehr schnell aus dem Speicher zu holen, bei Vektoroperationen zum Beispiel werden jedoch drei Speicherzugriffe pro Pipelinetakt erforderlich: 2 Eingabewerte und 1 Ergebniswert.

Die Zwischenschaltung lokaler Speicher soll diesen Engpaß beheben. Um den Faktor 10^4 - 10^5 sind die Zugriffe hier schneller als im Hauptspeicher, allerdings ist die Aufnahmekapazität solcher auch Puffer- oder Cachespeicher genannten Einheiten begrenzt.

Assoziativspeicher bilden eine mögliche Realisierungsform solcher lokalen Speicher. Hier wird gleichzeitig mit jedem Speicherwort auch seine Hauptspeicheradresse im Zwischenspeicher abgelegt. Wird ein Speicherwort in einem Befehl referiert, so wird zunächst mit seiner Adresse als Schlüssel im Zwischenspeicher nachgesehen, ob sich das Wort dort befindet. Ist dies der Fall, wird automatisch ein Lesezyklus gestartet, ist dies nicht der Fall, so muß mit der gewünschten Adresse ein Hauptspeicherzyklus initiiert werden. Wegen der hohen Kosten eines Assoziativspeichers wird dieses Verfahren in der Praxis jedoch recht selten angewandt.

Wesentlich häufiger werden schnelle Zwischenspeicher eingesetzt, die als "Swinging Buffer" oder "FIFO-Queues" organisiert sind (Bild 3.9).

Das Prinzip des Swinging Buffer besteht darin, daß zwei oder mehr schnelle RAM-Einheiten so miteinander verbunden sind, daß eine von ihnen vom Hauptspeicher geladen wird während eine andere die Pipeline mit Daten "beliefert". Sind beide Aktivitäten auf den entsprechenden Puffern beendet, so springen die Verbindungen um und der gerade geladene Puffer wird zur Pipeline geschaltet und umgekehrt. Das Abspeichern der Ergebnisse aus den arithmetischen Operationen der Pipeline gestaltet sich dann entsprechend.

Das FIFO-Verfahren (first in first out) ist eine recht einfache Struktur des Zwischenspeichers. Es werden zum Beispiel zwei Input-Queues vom Hauptspeicher geladen, bis nach einer gewissen Anfangszeit die Pipeline ihre benötigten Daten ausliest. Für die Abspeicherung der Ergebnisse gilt die Elementsprechend umgekehrt.

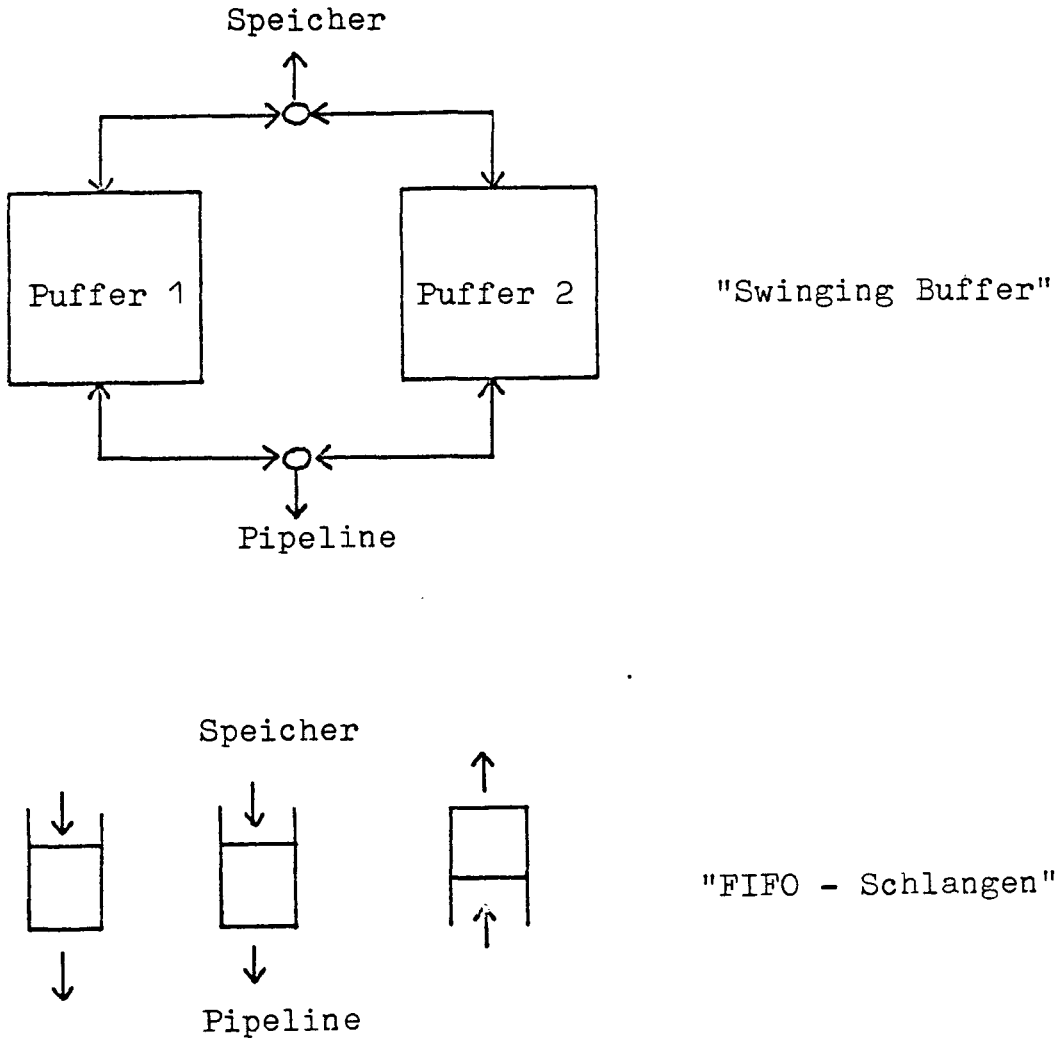


Bild 3.9: Beispiele für lokale Speicherorganisationen

Die Organisation der unterschiedlichen Speicherebenen, Hauptspeicher auf der einen und lokale Speicher auf der anderen Seite, trägt die wesentliche Verantwortung für eine möglichst optimale und damit effiziente Auslastung der Pipeline. Nicht minder wichtig ist die Busstruktur eines Rechners. Denn eine Hochgeschwindigkeitspipeline ist überflüssig, wenn keine Möglichkeit besteht, in entsprechender Zeit ausreichend viele Daten bereitzustellen.

3.3.4 Preis - Leistungsverhältnis einer Pipeline

Die Durchsatzrate D für eine Pipeline ergibt sich bei genügend großer Anzahl von durchzuführenden Operationen als

$$D \approx \frac{t}{N} \qquad (3.2)$$

mit t: Bearbeitungszeit für die gesamte Operation
N: Anzahl der Pipelinestufen.

Vorausgesetzt, die Bearbeitungszeit T für den Logikteil der Pipeline ist unabhängig von der Anzahl der Segmente, so läßt sich die Durchsatzrate verbessern, wenn mehr und dafür kürzere Pipelinestufen eingeführt werden. Diese Durchsatzrate D ist als wesentliches Maß für die Leistungsfähigkeit einer Pipeline anzusehen.

Neben der Leistung ist aber auch der Preis einer Pipeline von Interesse. Um eine brauchbare Kostenrelation angeben zu können, müssen jedoch die bei fortgesetzter Segmentierung zusätzlich erforderlichen Latches ebenfalls berücksichtigt werden. Es ergibt sich also bei konstanten Kosten C für die gesamte Logik einer Pipeline mit L als Kosten pro Latch eine Kostenfunktion K in Abhängigkeit von der Anzahl k der Pipelinestufen als(Bild 3.10a)

$$K = Lk + C \qquad (3.3)$$

Bezeichnet T die gesamte Logikzeit (im Unterschied zu t als Zeitbedarf für die gesamte Pipeline) und W den Zeitbedarf pro Latch, so ist der mittlere Zeitbedarf pro Pipelinestufe

$$\overline{T} = \frac{T}{k} + W \qquad (3.4)$$

und der Reziprokwert $1/\overline{T}$ gibt die Berechnungsrate der Pipeline an (Bild 3.10b). Das Preis - Leistungsverhältnis läßt sich dann darstellen als Quotient aus Kosten und Berechnungsrate (Bild 3.10c):

$$\begin{aligned} K \cdot \overline{T} &= (Lk + C)\left(\frac{T}{k} + W\right) \\ &= LT + LkW + \frac{CT}{k} + CW \\ &= LT + CW + \frac{Lk^2W + CT}{k} \end{aligned} \qquad (3.5)$$

Demnach ergibt sich der minimale Kostenaufwand pro Operation in Abhängigkeit von der Anzahl der Stufen als

$$\min_k (K \cdot T) = \sqrt{CT/LW} \qquad (3.6)$$

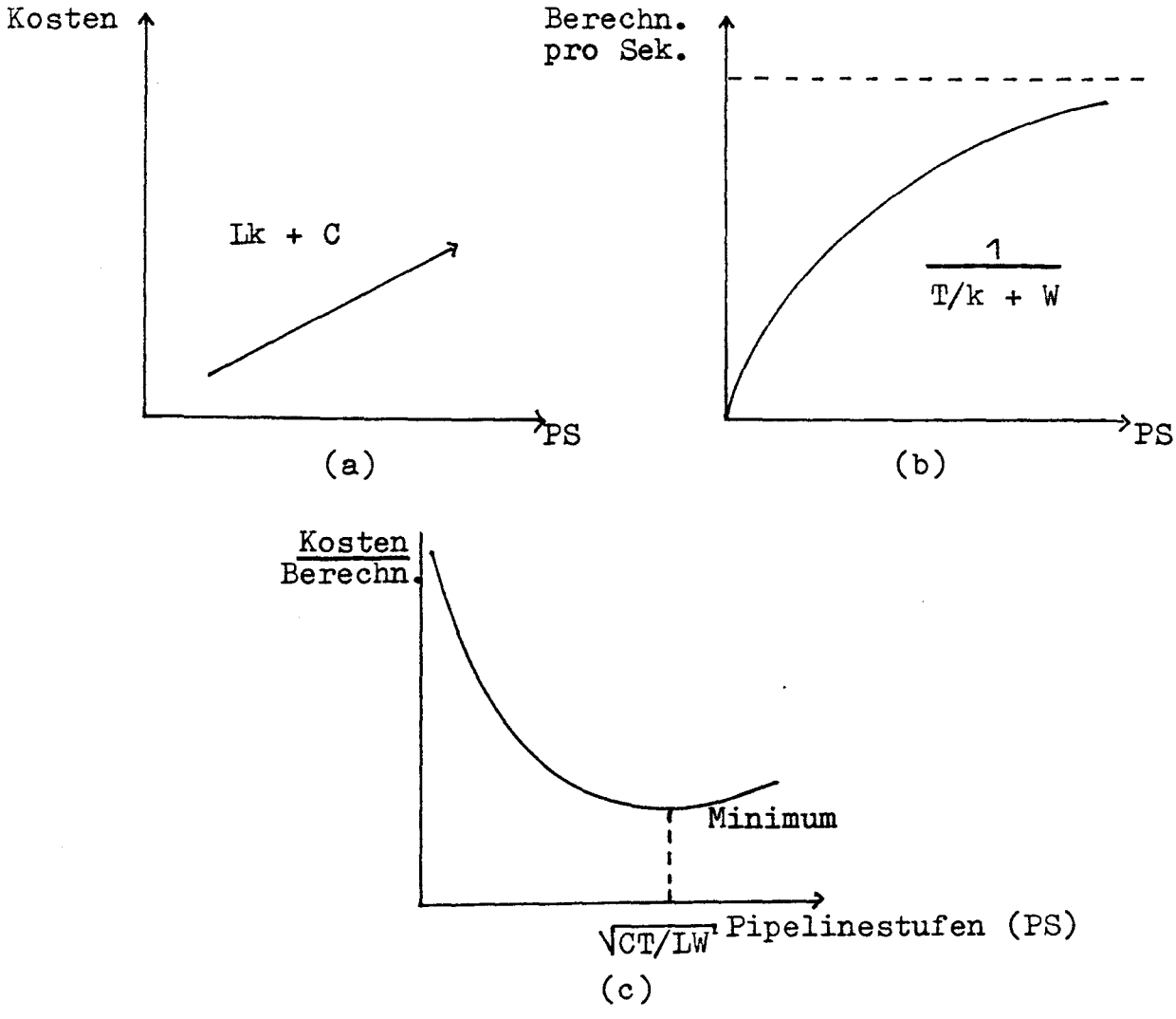


Bild 3.10: Preis - Leistungsverhältnis /Kog 81/

3.4 Die Kontrolle einer Pipeline

Nach der Übersicht über den Hardware-Aufbau einer Pipeline stellt sich die Frage nach der Kontrolle des Datenflusses innerhalb einer Pipeline. Dieses Problem gestaltet sich bei rein linearen Pipelines, das bedeutet, eine Stufe ist ausschließlich mit der ihr nachfolgenden verbunden (vgl. Bsp. in Kap. 2.4.4), recht problemlos. Die Daten werden rein sequentiell durchgeschoben, da jede Stufe nur ein einziges Mal während einer Berechnungsphase benutzt wird, und ohne großen Kontrollaufwand können die bereitgestellten Daten von der Pipeline übernommen und verarbeitet werden.

Bei komplexeren Pipelines besteht jedoch die Möglichkeit, daß Stufen mehrmals im Verlaufe einer Bearbeitung (Bild 3.11) benötigt werden. Diese Abweichung von der streng sequentiellen Abarbeitung kann vor allem bei dynamisch konfigurierten Pipelines auftreten.

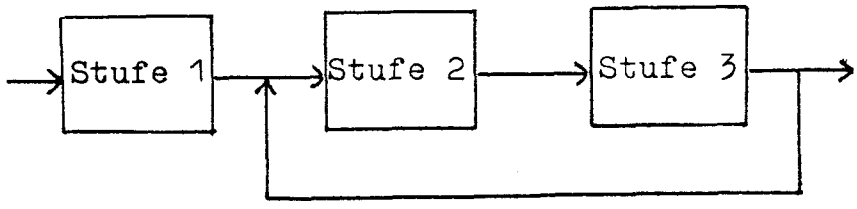


Bild 3.11: einfaches Beispiel für Mehrfachbenutzung bestimmter Pipelinestufen

Den Ablauf einer Berechnung kontrolliert jetzt der sogenannte Pipeline-Controller. Hierzu muß bei der Übernahme der Eingabedaten durch die Pipeline eindeutig feststehen, welche Stufe zu welchem Zeitpunkt für einen reibungslosen Berechnungsablauf bereitstehen muß. Diese Informationen verwaltet der Pipeline-Controller in einer "Reservierungstafel".

3.4.1 Die Reservierungstafel

Die Reservierungstafel ist im Prinzip bekannt als Raum-Zeit Diagramm für Pipelineverarbeitung aus Abschnitt 2.4.4. Sobald Daten ihren Weg durch die Pipeline beginnen, wird dieser Weg durch Eintragungen in die Reservierungstafel der entsprechenden Pipeline festgelegt. Für die Pipeline aus Figur 3.11 ergibt zum Beispiel ein solcher Eintragungsvorgang die Belegung in Bild 3.12.

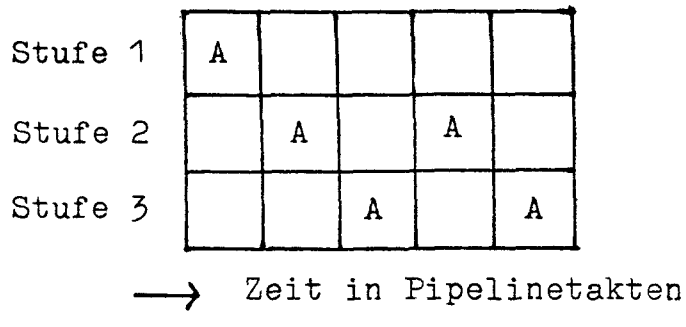


Bild 3.12: Reservierungstafel zur Berechnung von A mittels Pipeline aus Bild 3.11

Ein solcher Eintragungsvorgang zu Beginn einer Berechnung sei mit "Initiierung" bezeichnet.
Für statische Pipelines haben alle Initiierungen die gleiche Form, während bei dynamisch konfigurierten Pipelines unterschiedliche Initiierungen möglich sind. Im Folgenden soll zunächst ausschließlich die Kontrolle statischer Pipelines untersucht werden.

Soll nun pro Pipelinetakt eine neue Initiierung vorgenommen werden,so kommt es schnell zu Kollisionen, das heißt, unterschiedliche Berechnungen benötigen zur gleichen Zeit die gleiche Pipelinestufe (Bild 3.13).

Stufe 1	A	B	C	D				
Stufe 2		A	B	AC	BD	C	D	
Stufe 3			A	B	AC	BD	C	D

Bild 3.13: Reservierungstafel für statische Pipeline mit Kollisionen

Die Aufgabe des Pipeline-Controllers besteht nun darin, solche Kollisionen zu vermeiden, indem er die Initiierungen um eine bestimmte Anzahl von Pipelinetakten verschiebt. Für das Beispiel ergibt sich somit eine veränderte Reservierungstafel (ohne Kollisionen), wenn jede zweite Initiierung um zwei Takte verschoben wird (Bild 3.14).

1	A	B			C	D				
2		A	B	A	B	C	D	C	D	
3			A	B	A	B	C	D	C	D

Bild 3.14: Reservierungstafel ohne Kollisionen

Die Reihenfolge der Verschiebungen der einzelnen Initiierungen kann relativ zur zuletzt durchgeführten Initiierung angegeben werden, und für das Beispiel ergibt sich die Folge <1,2,1,2,1,2...>. Das heißt, eine mittlere Verschiebung um 1.5 Pipelinetakte ist vorzunehmen, um einen reibungslosen Berechnungsablauf zu garantieren.

3.4.2 Das Zustandsdiagramm

Das im vorigen Abschnitt in erster Linie durch Probieren erzielte Ergebnis läßt sich durch Einführen eines Kollisionsvektors sehr leicht schematisch nachvollziehen.

Ein Kollisionsvektor ist ein logischer Vektor und seine Länge ist bestimmt durch die Anzahl n der Takte pro Initiierung. Die Positionen dieses Vektors sind numeriert von 0 bis n-1. Die i-te Position erhält eine 1, wenn die Verschiebung einer Initiierung um i Takte eine Kollision

verursachen würde, eine 0, wenn sich keine Kollision ergibt. Eine Initiierung des mehrfach zitierten Beispiels (Bild 3.12) benötigt 5 Takte, und der entsprechende Kollisionsvektor hat die Form (10100).

Ein Zustandsdiagramm ist ein gerichteter Graph mit Kollisionsvektoren als Knoten und regelt eindeutig die Initiierungsfolge einer Pipeline.

Durch folgenden Algorithmus, 1971 von E. S. Davidson /Dav 71/ vorgestellt, läßt sich ein Zustandsdiagramm erzeugen:

0) Anfangszustand ist der Kollisionsvektor, der sich aus einer einzigen Initiierung ergibt.

1) Der aktuelle Kollisionsvektor wird, unter Verlust der nullten Position, nach links geschoben und eine 0 wird angehängt.

2) Ist die nullte Position jetzt gleich 1, dann ist dies der neue Kollisionsvektor. Gehe zu 1).

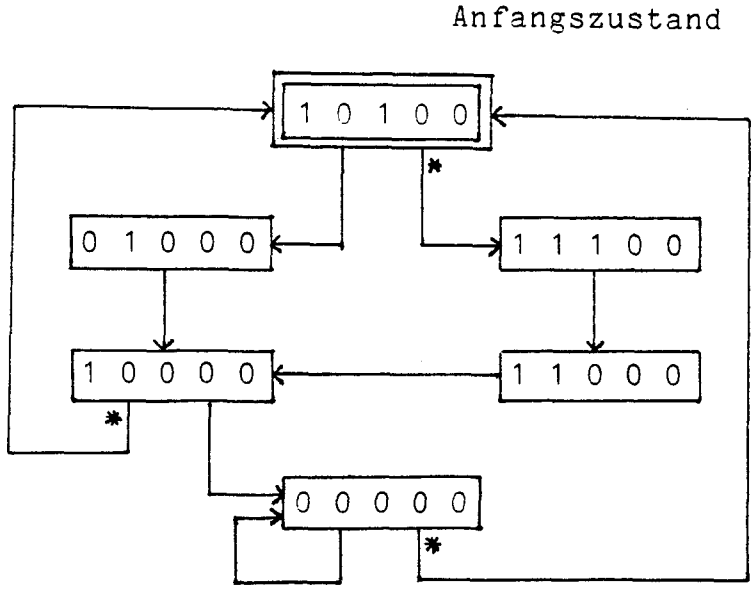
3) Ist die nullte Position gleich 0, dann bestehen zwei Möglichkeiten:

   a) eine neue Initiierung wird nicht vorgenommen:
      dann ist der vorliegende Vektor der neue Kollisionsvektor. Gehe zu 4).

   b) eine neue Initiierung wird vorgenommen:
      der aktuelle Vektor wird mit dem Ausgangsvektor bitweise durch logisches ODER verknüpft und der so gewonnene Vektor bildet den neuen Kollisionsvektor. Gehe zu 4).

4) Ist der ermittelte neue Kollisionsvektor bereits Element der Knotenmenge des Zustandsdiagramms, so erfolgt kein Hinzufügen eines neuen Zustandes, sondern eine Verbindung von Vorgänger und neu ermitteltem, aber vorhandenen Zustand durch eine gerichtete Kante. Ist der neu ermittelte Kollisionsvektor noch nicht Element der Knotenmenge, so wird dieser Vektor als neuer Zustand in das Zustandsdiagramm aufgenommen, indem eine gerichtete Kante vom Vorgänger zum neuen Zustand die Verbindung herstellt.

Das Zustandsdiagramm für die Pipeline aus Bild 3.11 ergibt sich demnach wie in Bild 3.15 gezeigt.



*: neue Initiierung wurde vorgenommen

Bild 3.15: Beispiel eines Zustandsdiagramms (für Pipeline aus Bild 3.11)

3.4.3 Kontrolle statischer Pipelines

Das Zustandsdiagramm ermöglicht dem Pipeline-Controller in Abhängigkeit vom momentanen Zustand der Pipeline, zu jedem Zeitpunkt zu entscheiden, ob im folgenden Pipelinetakt eine Initiierung erfolgen darf oder nicht.
Für das gezeigte Beispiel ist die Entwicklung des Zustandsdiagrammes und die damit mögliche Kontrolle des Datenflusses in einer Pipeline noch recht einfach. Bei komplexeren Pipelines und den sich daraus ergebenden längeren Kollisionsvektoren wird das entsprechende Zustandsdiagramm schnell sehr umfangreich, was eine effiziente Kontrolle der Pipeline beeinträchtigen kann.

Aus diesem Grunde führte Davidson auch ein modifiziertes Zustandsdiagramm ein. Der Unterschied zum ursprünglichen Diagramm kann in zwei Punkten zusammengefaßt werden:

1) Das modifizierte Zustandsdiagramm enthält nur solche Zustände, die sich unmittelbar aus Initiierungen ergeben, mit anderen Worten: die Zustände, die im ursprünglichen Diagramm einen mit einem Stern versehenen Eingangspfeil besitzen.

2) Zwei Zustände werden durch eine gerichtete Kante verbunden, wenn im ursprünglichen Diagramm eine Kantensequenz diese beiden Zustände verbindet, deren letzte Kante mit einem Stern versehen ist. Jeder dieser neuen Kanten wird eine Zahl zugeordnet, die sich aus der Anzahl der in der entsprechenden ursprünglichen Sequenz existierenden Kanten ergibt.

Den Umweg über das ursprüngliche Zustandsdiagramm kann man vermeiden, indem man das modifizierte Zustandsdiagramm direkt nach dem folgenden geänderten Algorithmus erzeugt.

Algorithmus zum modifizierten Zustandsdiagramm /Kog 81/:

0') = 0) Anfangszustand ist der Kollisionsvektor, der sich aus einer einzigen Initiierung ergibt.

1') Für jeden noch nicht verarbeiteten Zustand und für jedes k, so daß die k-te Position des entsprechenden Kollisionsvektors gleich 0 ist, führe aus
   a) Schieben um k Bits nach links unter Verlust der ersten k Bits und Anhängen von k Nullen
   b) neuer Zustand ergibt sich aus logischem ODER mit dem Anfangszustand
   c) Verbinden des neuen Zustands mit dem in Bearbeitung stehenden Zustand durch eine Kante mit dem Wert k

2') Sind alle vorhandenen Zustände bearbeitet: Einfügen von Kanten mit der Bezeichnung d von allen Zuständen zum Anfangszustand, mit d = Länge des Kollisionsvektors.

Für das Beispiel ergibt sich nach diesem Algorithmus ein modifiziertes Zustandsdiagramm wie in Bild 3.16a (vereinfacht: Bild 3.16b).

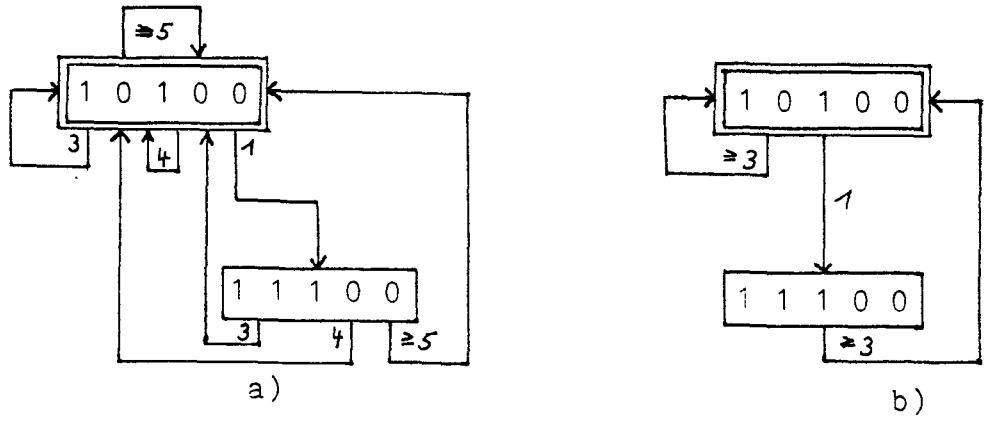


Bild 3.16: modifiziertes Zustandsdiagramm

Wesentlich deutlicher wird die Reduzierung des Aufwandes bei einer komplexeren Pipeline. Sei (110101101) der Kollisionsvektor für eine bestimmte Pipeline. Bild 3.17 zeigt die entsprechenden Zustandsdiagramme.

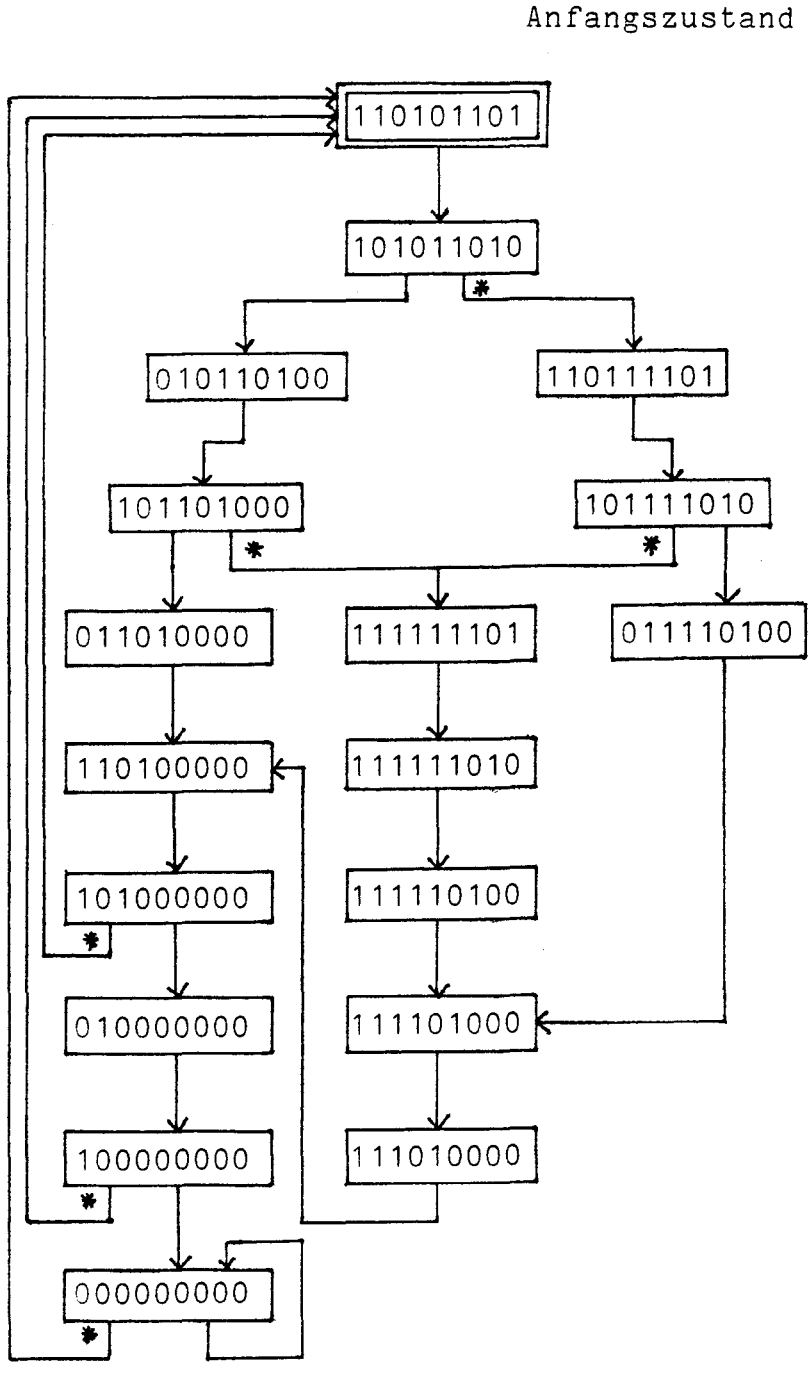


Bild 3.17a: einfaches Zustandsdiagramm

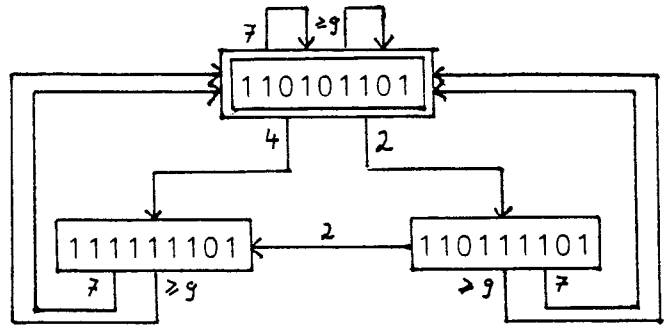


Bild 3.17b: modifiziertes Zustandsdiagramm

Die Zahlen an den Kanten des modifizierten Zustandsdiagramms entsprechen den Zeittakten einer Pipeline, um die eine Initiierung verschoben werden muß, damit Kollisionen entsprechend des aktuellen Zustands der Pipeline vermieden werden können.

Das Ziel der Verarbeitung von Daten durch eine Pipeline ist, einen möglichst hohen Durchsatz und damit eine große Bearbeitungsgeschwindigkeit zu erzielen. Demnach ist es sinnvoll, aus allen Zyklen eines modifizierten Zustandsdiagramms denjenigen auszuwählen, der die beste Initiierungsrate (Initiierungen/Takt) liefert.

Ein Zyklus ist ein Weg in einem Graphen über Kanten und Knoten, bei dem Anfangs- und Endknoten identisch sind. Dieser Weg kann daher beliebig oft hintereinander ausgeführt werden.

Tabelle 3.1 zeigt alle Zyklen aus dem Zustandsdiagramm in Bild 3.17b mit der entsprechenden Initiierungsrate. Die Zyklen werden angegeben mit den an ihren Kanten stehenden Zahlen.

Tabelle 3.1: Liste aller Zyklen aus Bild 3.17

Nr	Zyklus	Initiierungsrate	
1	(7)	1/7	
2	(9)	1/9	
3	(4,7)	2/11	
4	(4,9)	2/13	
5	(2,7)	2/9	
6	(2,9)	2/11	
7	(2,2,7)	3/11	← max
8	(2,2,9)	3/13	

Der Zyklus Nr. 7 liefert die maximale Initiierungsrate. Demnach läßt sich der höchst mögliche Durchsatz ohne Kollisionen bei Initiierungen in den Pipelinetakten 0,2,3,11,13,15,22,24,26,33,... erreichen.

Nach einem Satz von L. E. Shar /Sha 72,Kog 81/ kann die Suche nach solchen optimalen Zyklen auf einfache Zyklen beschränkt werden, da es zu jedem Zyklus mit der Initiierungsrate I mindestens einen einfachen Zyklus mit einer Initiierungsrate I gibt.

Als einfach werden solche Zyklen bezeichnet, die jeden ihrer Knoten nur ein einziges Mal während eines Durchlaufs berühren. Die Folge des obigen Satzes ist eine wesentliche Vereinfachung der Suche nach optimalen Zyklen.

Die Angabe eines wichtigen und interessanten Zusammenhangs zwischen der mittleren Verschiebung V (Initiierungsrate^{-1}) eines optimalen einfachen Zyklus und der Anzahl n der 1-Positionen im Anfangszustand des Zustandsdiagramms gelang ebenfalls Shar mit dem Beweis, daß gilt:

$$V \leq n.$$

Die Anzahl der 1-Positionen ist also eine obere Grenze für die durchschnittlich notwendigen Verschiebungen von Initiierungen zur Vermeidung von Kollisionen.
Das Verfahren zur Suche nach einem optimalen Zyklus im modifizierten Zustandsdiagramm ist natürlich nur dann sinnvoll, wenn eine genügend große Zahl von Daten von der Pipeline zu bearbeiten ist. Denn sonst ist es möglich, daß der Aufwand hierfür selbst die Bearbeitungszeit für einen nicht optimalen Zyklus übersteigen kann.

3.4.4 Kontrolle dynamischer Pipelines

Dynamische Pipelines zeichnen sich dadurch aus, daß die zu berechnende Funktion häufig geändert wird. Im Gegensatz zu statischen Pipelines, bei denen Initiierungen jeweils gemäß ein und derselben Reservierungstafel vorgenommen werden, ist im dynamischen Fall eine bestimmte Anzahl unterschiedlicher Reservierungstafeln mit möglicherweise unterschiedlichen Längen als Initiierungsvorlage zugelassen. Dies kann dazu führen, daß zwei oder mehr Initiierungen in einem Pipelinetakt vorgenommen werden können, wenn sie in jedem Takt verschiedene Pipelinestufen benötigen.

Hier wird bereits deutlich, daß die Kontrolle des Datenflusses in einer dynamischen Pipeline ungleich aufwendiger als im statischen Falle ist. Zunächst einmal müssen die Kollisionsvektoren durch Kollisionsmatrizen CM ersetzt werden. Sie haben die Größe r x d mit:

r: Anzahl der möglichen verschiedenen Reservierungstafeln
d:= $\max_i$ {Länge des Kollisionsvektors zu Reservierungstafel i}

Diese binären Kollisionsmatrizen bilden jetzt die Zustände des Zustandsdiagramms.

Ein weiterer Unterschied zum Zustandsdiagramm für statische Pipelines besteht darin, daß r verschiedene Anfangszustände CM existieren. Die j-te Zeile im Anfangszustand CM ist der Kollisionsvektor, der sich ergibt aus einer durchgeführten Initiierung gemäß Reservierungstafel i und einer gleichzeitigen oder späteren Initiierung gemäß Reservierungstafel j. Für jede Reservierungstafel müssen also alle möglichen Nachfolgekonfigurationen berücksichtigt werden. Daraus resultieren die r Anfangszustände.

Die Matrix CM erhält in der Position (j,k) eine 0, falls k Pipelinetakte nach einer Initiierung vom Typ i eine Initiierung vom Typ j kollisionsfrei durchgeführt werden kann, sonst eine 1.

Ausgehend von der Reservierungstafel Bild 3.10 und zwei weiteren Tafeln (Bild 3.18) ergeben sich somit die folgenden drei Matrizen als Anfangszustände eines Zustandsdiagramms für eine dynamische Pipeline:

$$CM_A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \qquad CM_B = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

$$CM_C = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

1		B		B		B
2	B				B	
3			B	B		

1			C	
2		C		C
3	C			

Bild 3.18: Reservierungstafeln für unterschiedliche Berechnungen auf einer 3-stufigen Pipeline

Die Entwicklung des Zustandsdiagramms läuft ab wie im statischen Fall, allerdings ist jetzt zu beachten:

- jeweils die ganze Matrix wird nach links geschoben und in jeder Zeile eine 0 pro Schiebetakt angehängt
- die i-te so entstandene Zeile wird mit der i-ten Zeile des entsprechenden Anfangszustandes durch logisches ODER verknüpft für alle i mit $1 \leq i \leq r$.

- die so entstandene Matrix wird als neuer Zustand aufgenommen, wenn sie nicht bereits im Zustandsdiagramm existiert, und wie im statischen Fall tragen die Verbindungskanten die Anzahl der durchgeführten Schiebetakte und die Kennzeichnung einer erfolgten Initiierung.

Tabelle 3.2 gibt einen Eindruck von der Komplexität eines Zustandsdiagramms für dynamische Pipelines. Hier sind ausschließlich die direkten Nachfolgezustände des Anfangszustandes CM im modifizierten Diagramm aufgelistet.

Tabelle 3.2: direkte Nachfolger von $CM_B = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$

Anzahl der Schiebetakte	Initiierung	Zustandsmatrix
0	C	$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$
2	A	$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$
3	B	$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$
4	A,AC	$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$
4	C	$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$
≥5	B	$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix} = CM_B$
≥5	C	$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} = CM_C$
≥6	A	$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} = CM_A$

Eine allgemeingültige Strategie zur Suche eines optimalen Zyklus in einem Zustandsdiagramm für dynamische Pipelines kann nicht angegeben werden, da sich verschiedene Optimierungskriterien ansetzen lassen.

Je nach Problem ist zu entscheiden, ob

a) die Gesamtzahl beliebiger Initiierungen maximiert werden soll,

b) die Gesamtzahl von Initiierungen unter Berücksichtigung eines bestimmten Anteils der einzelnen Reservierungstafeln maximiert werden soll,

c) die Zeit für die Bearbeitung einer fest vorgegebenen Initiierungsfolge minimiert werden soll.

Für jedes dieser Kriterien kann sich ein anderer Zyklus als optimal herausstellen. Wird a) als Kriterium angesetzt, so läßt sich eine Lösung noch nach dem in Abschnitt 3.4.3 angegebenen Verfahren für statische Pipelines finden. Für b) und c) sind jedoch schon aufwendigere Branch-and-Bound Algorithmen sowie die Lineare Programmierung als Teilbereiche aus dem Operations Research anzuwenden.

Der STAR 100 von CDC ist ein Beispiel eines Rechners mit komplexen Pipeline-Funktionseinheiten. Hier ließe sich eine der gezeigten Kontrollmöglichkeiten anwenden. Der STAR 100 besteht im wesentlichen aus einem stark verschränkten Kernspeicher, einer "streaming unit", die den Datenfluss zwischen Hauptspeicher und Pipelines regelt, und zwei Pipeline-Prozessoren. Diese Pipeline-Prozessoren sind statisch konfigurierte Mehrfunktionspipelines (Bild 3.19).

Die Zahl der Verarbeitungsstufen hängt ab von der jeweils auszuführenden Operation. Die Maximalzahl liegt bei etwa 30. Der Takt eines elementaren Verarbeitungsschrittes beträgt 40 ns. Allerdings muß eine hohe Startzeit der Pipelines von bis zu 3000 ns /Gil 81/ in Kauf genommen werden.

Jeder der beiden Pipeline-Prozessoren beinhaltet einen 6-stufigen Gleitpunktaddierer. Prozessor 1 verfügt außerdem über zwei Gleitpunktmultiplizierer und einen speziellen Addierer, der ein 64-Bit-Produkt aus zwei 32-Bit-Teilprodukten zusammensetzen kann. Prozessor 2 besitzt noch ein Mehrfunktionsrechenwerk für Gleitpunktmultiplikation, -division und -wurzelziehen. Jede dieser Teileinheiten ist wiederum als Pipeline organisiert. Ein weiterer Dividierer ist als einziger Teil nicht segmentiert.

Zum Schluß dieses Abschnittes bleibt zu vermerken, daß zur Zeit solche komplexen Pipeline-Konstruktionen noch nicht sehr verbreitet sind. Die meisten modernen Pipelinerechner besitzen rein sequentiell aufgebaute Pipeline-Funktionseinheiten.

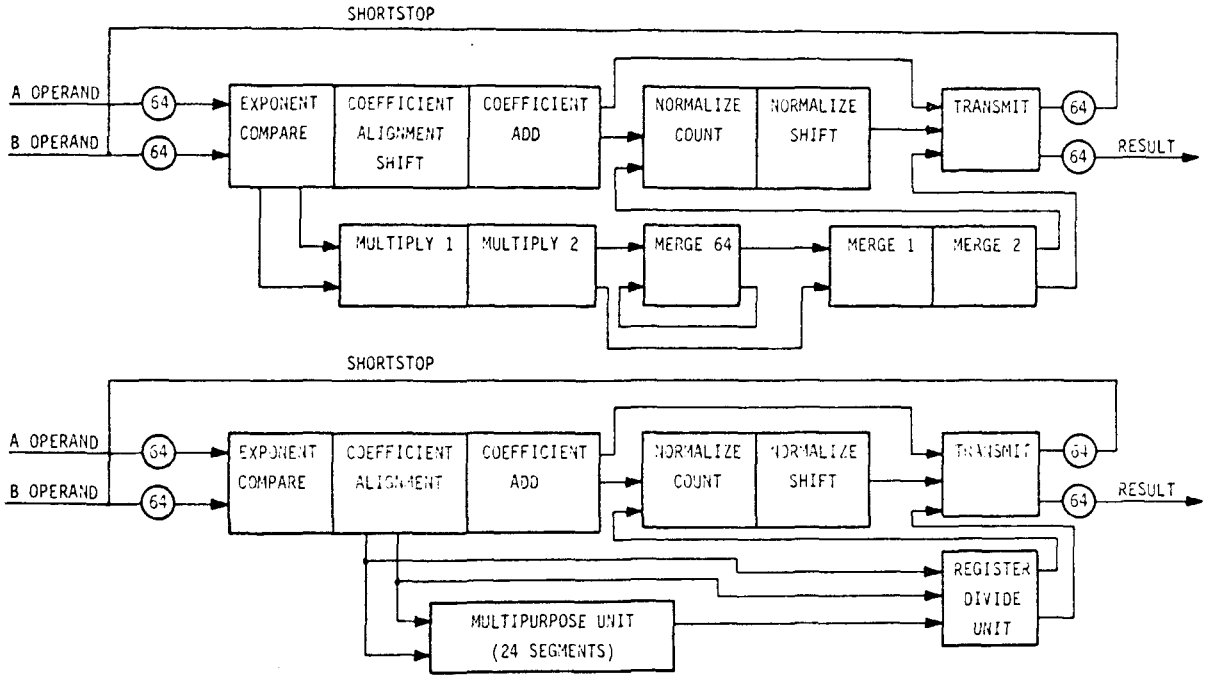


Bild 3.19: Pipeline-Prozessoren des STAR 100 /Hat 72/

3.5 Beispiele moderner Pipeline-Rechner

3.5.1 Cray Research CRAY-1

Die CRAY-1 /Rus 78/ ist ein Großrechner, der konzipiert wurde zur Ausführung von Hochgeschwindigkeitsvektoroperationen. Mit einer Taktzeit von nur 12.5 ns und einer Rechenleistung von bis zu 240 MFLOPS (million floating-point operations per second) - allerdings nur für kurze Rechenperioden - nimmt dieser Rechner eine führende Position unter den modernen Hochgeschwindigkeitsrechnern ein.

Die extremen Geschwindigkeiten werden erreicht durch das Zusammenspiel von drei unterschiedlichen Faktoren:

a) Fortschritte in der Bauteiltechnologie (z.B.: 50 ns Halbleiterspeicher)

b) relativ geringe Baugröße, dadurch keine Verzögerungen aufgrund der Weglänge, die die elektrischen Signale zurüchzulegen haben

c) Pipeline-Funktionseinheiten mit einem hohen Grad an Parallelität in der Ausführung der Operationen

Mit der Außenwelt ist die CRAY-1 über einen Front/Endrechner, der die E/A-Funktionen übernimmt, verbunden. Die wichtigsten architektonischen Merkmale sind im Folgenden zusammengefaßt.

Die arithmetischen und logischen Berechnungen werden in 12 Funktionseinheiten (Bild 3.20) durchgeführt, die jeweils als Pipeline realisiert sind. Sechs von diesen sind ausschließlich dem Instruktions-Prozessor unterstellt, der alle Kontrollfunktionen sowie die Befehlsinterpretation durchführt. Die entsprechenden Funktionseinheiten werden zur Adreßrechnung, zur Skalar - Integer Rechnung und für logische Operationen herangezogen.

Die eigentliche Datenbearbeitung erfolgt in drei Einheiten für die Gleitpunktarithmetik sowie in drei Einheiten für die Vektor - Integer Rechnung.
Die Funktion der lokalen Speicher übernehmen neben einigen Adreß- und Skalarregistern im Instruktions-Prozessor im wesentlichen 8 Vektorregister zu je 64 Vektorelementen. Ausschließlich in diesen Registern werden die zu bearbeitenden Vektoren für die arithmetischen Pipelines bereitgestellt.

Wird zum Beispiel eine Vektor - Vektor Addition durchgeführt, so sind zwei Register für die Eingabe verantwortlich und ein drittes bleibt bis zum Ende der Operation für das Zwischenspeichern der Ergebnisse reserviert. Dies stellt eine Blockierung dar, kann jedoch durch das sogenannte "vector-chaining" umgangen werden. Das bedeutet, unter bestimmten zeitlichen Voraussetzungen kann das Ergebnisregister einer noch nicht beendeten Operation bereits als Eingaberegister für eine andere Funktionseinheit verwendet werden.

Die Länge des zu bearbeitenden Vektors wird im VL-Register (vector length) überwacht. Das VM-Register (vector mask) ermöglicht die Beschränkung einer Vektoroperation auf einen Teil der Elemente eines Vektors.

Ist eine Vektoroperation beendet bzw. das Ergebnisregister gefüllt, so werden die Ergebnisse in den Hauptspeicher zurückgeschrieben. Dieser Hauptspeicher besteht aus 16 Speicherbänken zu je 72 Moduln. Jeder dieser Moduln beinhaltet jeweils 1 Bit eines Speicherwortes - 64 Bits pro Wort + 8 Kontrollbits für SECDED (single error correction, double error detection).

Die Ausführung der Operationen auf Registerinhalten erfordert eine Zerlegung der zu bearbeitenden Vektoren in Blöcke der Länge 64, hat aber den Vorteil, daß bei kurzer Vektorlänge oder bei Skalaroperationen die Leistung nicht stark absinken kann, da die Aufsetzzeiten für die Pipelines nicht so sehr ins Gewicht fallen.

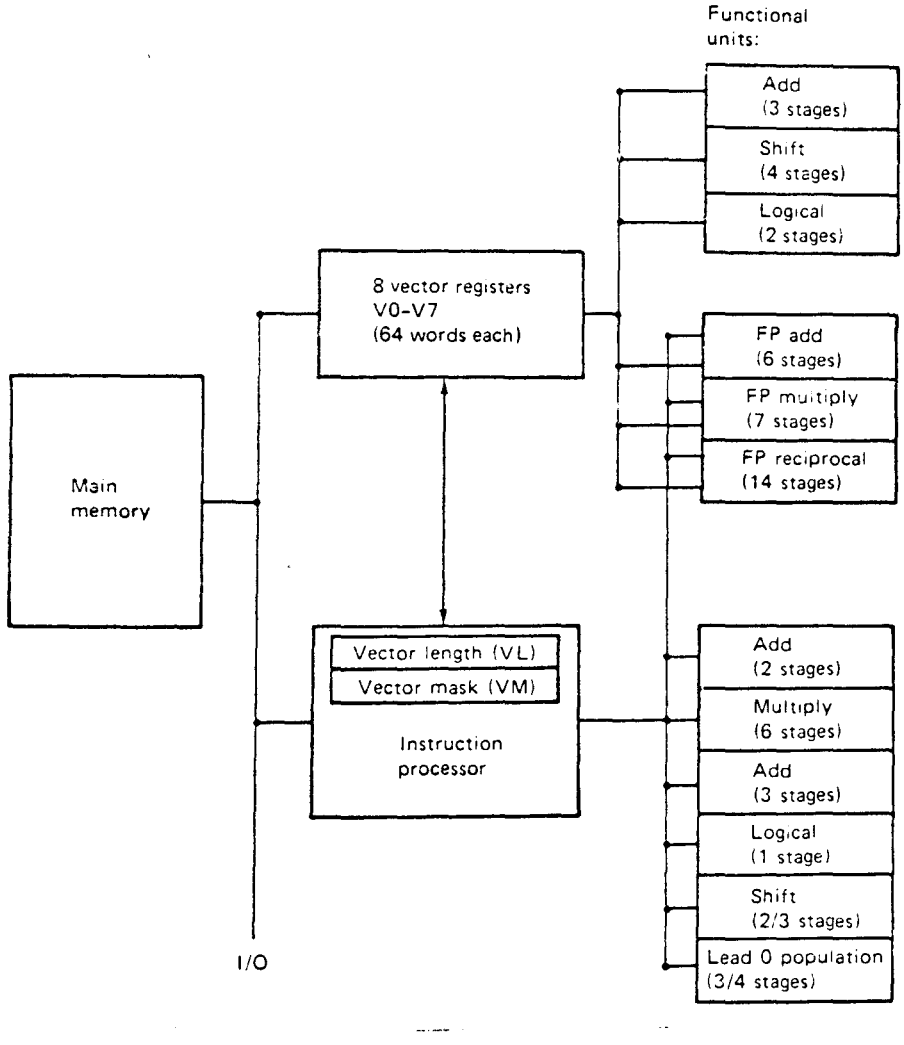


Bild 3.20: Blockschema der CRAY-1 /Kog 81/

3.5.2 CYBER 205

Die CYBER 205 /Cyb 82/ ist einer der neuesten Hochgeschwindigkeits-Großrechner auf dem Markt. Das System wurde ausschließlich für Rechenzwecke konzipiert und gilt als Nachfolger des STAR 100 und als Weiterentwicklung des direkten Vorgängers CYBER 203. Auch hier wird für E/A-Tätigkeit und Verwaltungsaufgaben ein Front-/Endrechner eingesetzt.
Neben einer Wartungseinheit und dem Hauptspeicher - 1 bis 4 Millionen 64-Bit Datenworte - bildet das Kernstück die aus zwei im wesentlichen unabhängig voneinander arbeitenden Prozessoren bestehende CPU (Bild 3.21).

Der Skalarprozessor ist neben seiner Funktion als Rechner für skalare Größen die steuernde Einheit des Systems, erledigt Instruktionsdekodierung, Speicherzugriffe, virtuelle Adressierung und ähnliches.

Der Vektorprozessor hat je nach Ausführung eine, zwei oder vier Gleitpunktpipelines für Vektoroperationen und eine "string-unit" für Bitkettenoperationen. Jede 64-Bit Pipeline entspricht wiederum zwei 32-Bit Pipelines, so daß der Durchsatz bei halber Genauigkeit verdoppelt werden kann.
Eine Pipeline (64-Bit) besitzt einen Datenaustauschteil, welcher die Daten für die arithmetischen und logischen Operationen bereitstellt und als Zwischenspeicher für die Ergebnisse auf dem Weg zum Speicher dient (lokaler Speicher).

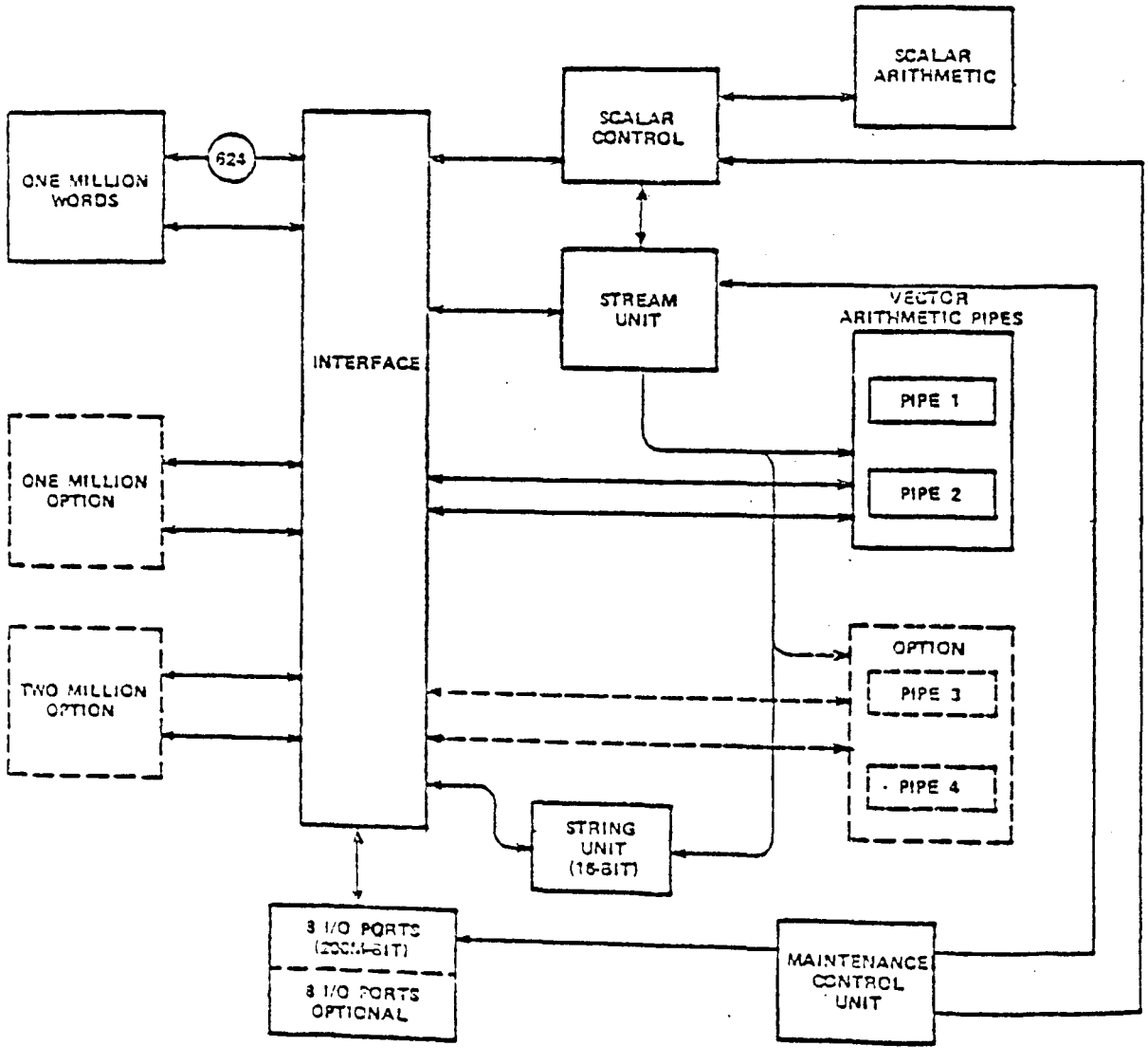


Bild 3.21: Blockbild der CYBER 205

Die Berechnungen können jeweils in einer von 5 ALUs ("ADD","MULT","SHIFT","LOGICAL","DELAY") ausgeführt werden. In jedem Fall bearbeiten alle Pipelines die gleiche Vektoroperation und die Komponenten werden abwechselnd auf die einzelnen Pipelines verteilt.

Die Verarbeitung langer Datenvektoren erfolgt also nicht, wie bei der CRAY-1, in Blöcken bestimmter Länge, sondern die gesamten Vektoren werden in einem Datenstrom vom Hauptspeicher über einige Puffer in die Pipelines geleitet, und die Ergebnisse kommen auf dem umgekehrten Weg in den Speicher zurück.

Der Hauptspeicher besteht aus 16 Moduln, die jeweils in 8 Bänke unterteilt sind. Jedes Speicherwort umfaßt 64 Datenbits + je 7 SECDED-Bits pro Halbwort, also insgesamt 78 Bit.
Aufgrund der Speicherverschränkung kann pro Takt, der für die CYBER 205 20 ns beträgt, ein Speicherzugriff erfolgen. Die Taktzeit ist zwar länger als bei der CRAY-1, die maximale Rechenleistung wird jedoch mit bis zu 800 MFLOPS angegeben, womit die CYBER 205 der derzeit schnellste Hochgeschwindigkeitspipeliner ist. Allerdings ist dieser Wert nur bei halber Genauigkeit der Gleitpunktoperationen zu erreichen.

3.5.3 IBM 3838 Array Processor

Der IBM 3838 Array Processor /IBM 76,Kog 81/ ist,wie sein Vorgänger IBM 2938, ein spezieller Rechner zur Verarbeitung rechenintensiver Teilprogramme. Das gesamte Programm wird von einem sogenannten Host-Rechner ausgeführt, der nur die entsprechenden Teilprogramme sowie die dazu nötigen Daten über einen E/A-Kanal dem 3838 AP übermittelt. Dieser führt dann, einmal angestoßen, parallel zum Host-Rechner die rechenintensiven Vorgänge aus.

Hauptkomponente des 3838 AP ist die arithmetische Einheit (Bild 3.22). Sie umfaßt einen Multiplizierer und zwei Addierer für Geitpunktoperationen, die jeweils als 4-stufige Pipeline realisiert sind. Auch Reziprokwert-Einheit, Sinus/Cosinus-Generator und die Kontrolleinheit sind nach dem Pipeline-Prinzip in unterschiedliche Anzahlen von Segmenten aufgeteilt.

Zwei Speicherbänke zu je 8K Byte erledigen die Funktion des lokalen Speichers und sind als Swinging Buffer organisiert. Zwischen diesem Arbeitsspeicher und dem Hauptspeicher unterliegt die Datenübermittlung der Kontrolle des Data Transfer Controllers (DTC).

Die Taktzeit pro Pipelinestufe beträgt 100 ns. Ein Arbeitsspeicherzyklus benötigt 400 ns, ein Hauptspeicherzyklus 800 ns. Durch komplexe Verschränkung ist jedoch eine maximale Datenrate von 40 Megabyte /IBM 76/ zu erreichen.

Bis zu sieben Benutzer können jeweils gleichzeitig vom 3838 AP "bedient" werden, da die Teileinheiten wie E/A-Kanal, arithmetische Einheit, DTC usw. vollständig unabhängig voneinander arbeiten.

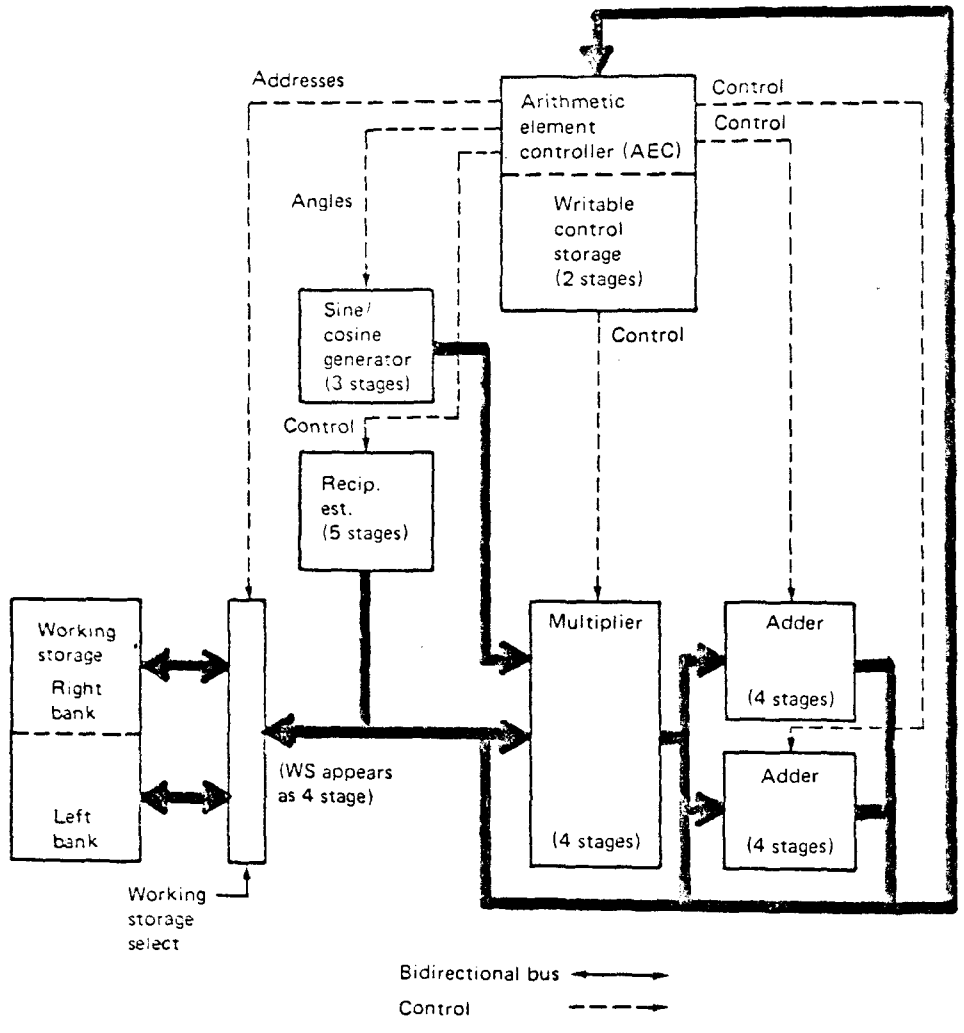


Bild 3.22: Arithmetische Einheit und Arbeitsspeicher des IBM 3838 AP /Kog 81/

3.5.4 FPS AP190L

Der Array Processor der Firma Floating Point Systems, Portland/Oregon, ist wie der IBM 3838 AP ein peripherer Rechner für Gleitpunktarithmetik, der parallel zu einem Host-Rechner rechenintensive Teilprogramme bearbeiten kann. Im Folgenden wird dieser, kurz mit AP bezeichnete Rechner, etwas ausführlicher als die bisherigen Beispiele behandelt, da in Kapitel 4 "Programmierung" einige spezielle Probleme für die effiziente Lösung auf einem AP untersucht werden.

Bild 3.23 zeigt die einzelnen Elemente des AP und seine hochgradig parallele Busstruktur.

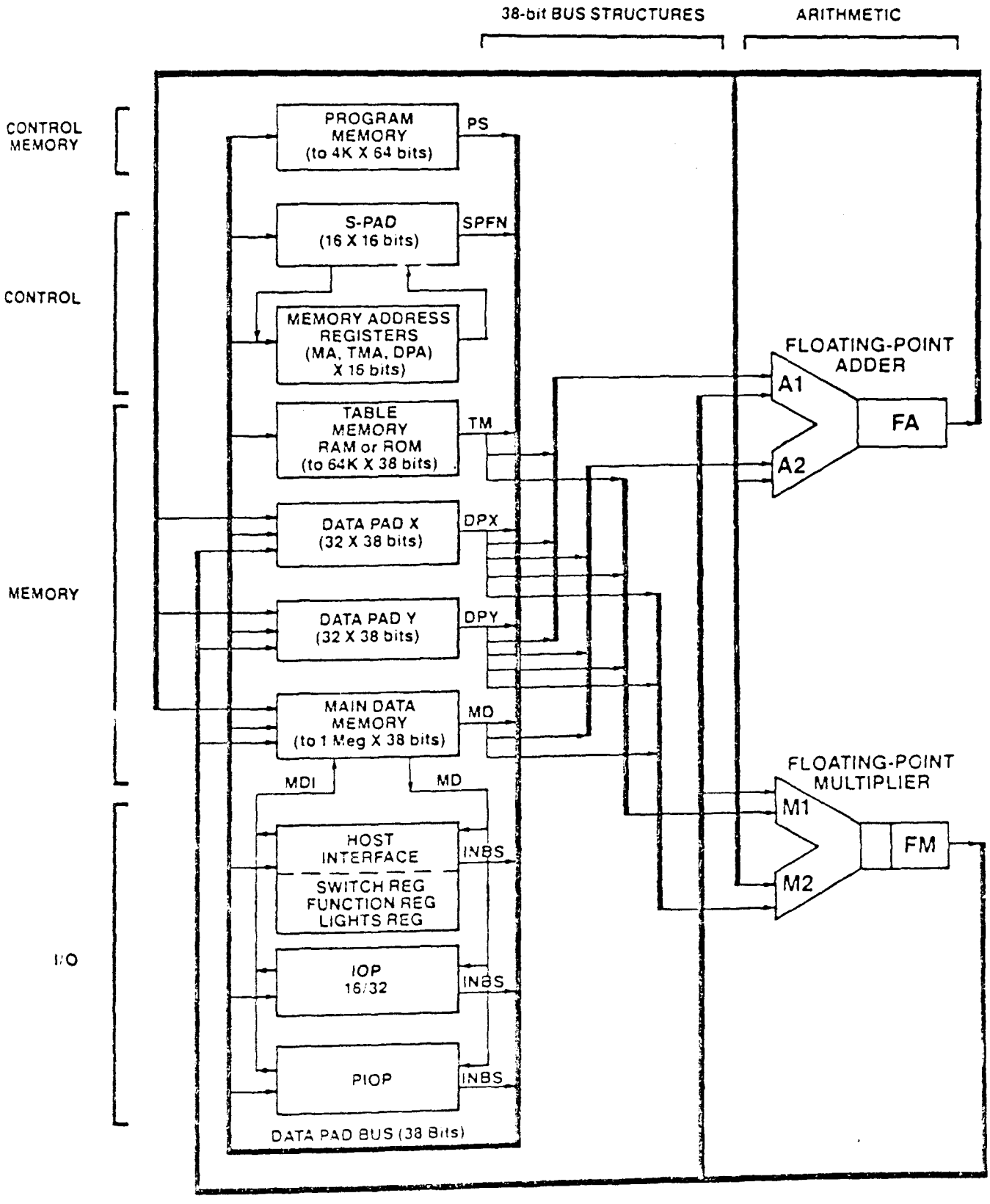


Bild 3.23: Busstruktur und Elemente des FPS AP190L

Die Maschine ist insgesamt wesentlich kleiner als die bisher gezeigten Beispiele und arbeitet mit einer Taktzeit von "nur" 167 ns.

Die Länge für die im Programmspeicher abzulegenden Befehlsworte beträgt 64 Bit. 4K Worte umfaßt dieser Programmspeicher. Jedes der Befehlsworte ist in 10 Befehlsfelder unterteilt, deren Inhalte alle in einer Taktzeit abgearbeitet werden.

Durch das S - PAD erfolgt die Steuerung des Ablaufs innerhalb eines Programms. 16 Integer Register und eine ALU für Integer-Arithmetik stehen dort für Adreßrechnung, Schleifenzählung usw. zur Verfügung.

Die internen Datenworte haben eine Länge von 38 Bit (28 Bit Mantisse + 10 Bit Exponent) und werden während des Transfers vom Host-Rechner zum AP auf diese Größe umgeformt ("conversion on the fly"). Die zu bearbeitenden Daten werden vor Beginn einer Berechnung vom Host-Rechner in den Datenspeicher des AP geladen. Dieser Datenspeicher ist im vorliegenden Modell vierfach verschränkt, und zwar sind die Speicherbänke wie folgt aufgeteilt:

1. Bank: Adressen 0,2,4,...,32766
2. Bank: Adressen 1,3,5,...,32767
3. Bank: Adressen 32768,32770,...,65534
4. Bank: Adressen 32769,32771,...,65535

Zu jedem zweiten Takt kann dieselbe Speicherbank angesprochen werden, und es vergehen 3 Takte, bis der Inhalt der angesprochenen Speicherzelle im Memory Destination Register (MD) zur Verfügung steht. Sind die Daten abwechselnd auf geraden und ungeraden Adressen abgelegt, so kann in jedem Takt ein Zugriff (abwechselnd auf zwei Bänke) durchgeführt werden. Wird dies nicht eingehalten, so werden intern Zugriffskonflikte mit Hilfe eines Wartezyklus gelöst.

Der Tabellenspeicher besteht aus zwei Teilen: 2.5 K Worte ROM enthalten bestimmte Konstanten und 2 K Worte RAM können auch vom Benutzer genutzt werden.

Data Pad X und Data Pad Y sind Registerblöcke zu je 32 Registern und stehen als Arbeitsregister für die Gleitpunkt-Arithmetik zur Verfügung. Wichtig ist hier, daß in einem Takt in jedem Registerblock ein Register beschrieben und ein anderes gelesen werden kann. Auch Lesen und Beschreiben desselben Registers in einer Taktzeit ist möglich nach dem Prinzip "write after read".

Beide Registerblocks besitzen einen gemeinsamen Pointer (DPA = Data Pad Address Register). Direkter Zugriff ist jeweils nur auf 8 Register möglich, die relativ zum Pointer (-4 bis +3) adressiert werden können. Um Register außerhalb dieses Bereiches ansprechen zu können, ist das Umsetzen des Pointers erforderlich.

Die Gleitpunkt-Arithmetik wird von zwei als Pipelines konstruierten Funktionseinheiten durchgeführt. Ein 3-stufiger Multiplizierer mit den Eingaberegistern M1 und M2 sowie ein 2-stufiger Addierer mit A1 und A2 als Eingaberegister können parallel in einem Arbeitstakt von 167 ns arbeiten.

Beide Funktionseinheiten benötigen explizite Instruktionen - FADD bzw. FMUL -, um die Zwischenergebnisse innerhalb der Pipeline weiter zu schieben oder die Resultate am Ende einer Pipeline im FA- oder FM-Register verfügbar zu machen.

Lassen Problemstellung und Struktur des AP es zu, daß beide Funktionseinheiten pro Takt ein neues Operandenpaar erhalten, so kann eine maximale Rechenleistung von 12 MFLOPS erreicht werden (pro Funktionseinheit 6 MFLOPS). Diese Rechenleistung wird ermöglicht durch den hohen Grad an Parallelität in der Struktur des AP. So können zum Beispiel Index- und Adreßrechnungen, Speicherzyklen und arithmetische Operationen parallel ausgeführt werden.

Allerdings ist zu beachten, daß die maximale Rechenleistung nur dann erreichbar ist, wenn die benötigten Daten auch rechtzeitig, das heißt pro Takt, den beiden Pipelines zur Verfügung gestellt werden können. Dies ist wie auch in Kapitel 4 noch zu sehen sein wird, nur bei speziellen Problemen möglich.

Als letzter Punkt in dieser kurzen Übersicht über den AP bleibt noch ein Wort zum I/O-Komplex zu sagen. Diese I/O-Einheiten stellen ausschließlich die Verbindungen zwischen Host-Rechner und AP her und führen gleichzeitig zum Datentransfer die Konvertierung der Daten in die entsprechenden AP-Formate durch. Der AP besitzt kein eigenes Betriebssystem, daher unterliegt der Programmablauf der Kontrolle des Host-Rechners.

Entscheidend für eine bestmögliche Rechenleistung ist in allen Fällen eine Programmierung unter Berücksichtigung der parallelen Möglichkeiten, die ein Rechnersystem dem Benutzer bietet. Um die Hardware jedoch effizient nutzen zu können muß auf die Programmierung in einer Assembler-Sprache zurückgegriffen werden.

Kapitel 4 befaßt sich mit der Problematik bei der Programmierung von Pipelinerechnern und untersucht die Lösungsmöglichkeiten für spezielle Probleme auf dem hier vorgestellten FPS AP190L.

4. Programmierung

4.1 Einführung

Für den Begriff "Programmierung" ergibt sich im allgemeinen keine derart breit gefächerte Definitionsskala, wie sie für die Architektur in Kapitel 2.1 aufgezeigt wurde. Vielmehr wird unter Programmierung in erster Linie die Formulierung bzw. Codierung eines vorher analysierten Problems in einer dem vorliegenden Rechner verständlichen Sprache verstanden.

Der Weg eines Problems bis zu seiner Lösung im Rechner gliedert sich im wesentlichen in drei Teilschritte, die die Darstellung des Problems jeweils verändern (Bild 4.1):

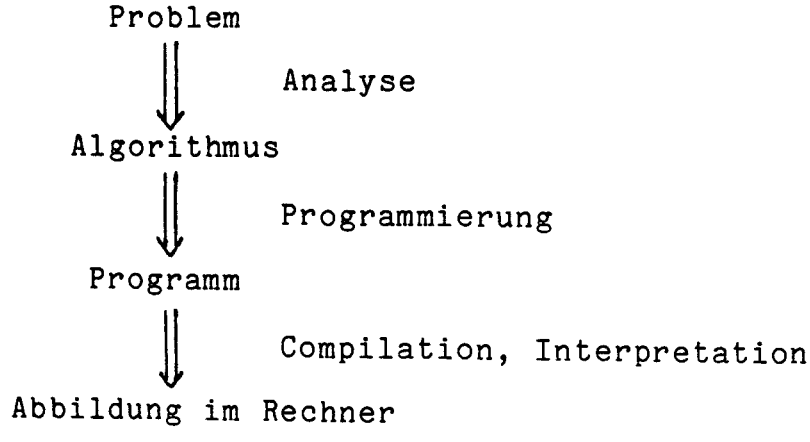


Bild 4.1: Darstellungsformen eines Problems

Für alle innovativen Architekturformen und insbesondere für Pipelinerechner gilt allerdings, daß die Programmierung und die aus der Problemanalyse resultierende Algorithmenentwicklung nicht getrennt voneinander gesehen werden können. Vielmehr sind sie eng miteinander verbunden und beeinflussen sich, zumindest teilweise, gegenseitig.

Denn wesentlich wichtiger als bei der Programmierung des konventionellen von-Neumann Rechners sind jetzt die durch das architektonische Konzept des jeweiligen Rechners beeinflußten programmiertechnischen Möglichkeiten, zum Beispiel parallele Nutzung unterschiedlicher Hardwareelemente, die die Entwicklung effizienter Algorithmen entscheidend prägen müssen, um dann schnellst mögliche Lösungen zu erzielen. Hierbei ist zu berücksichtigen, daß die optimale Ausnutzung der architektonischen Gegebenheiten zur Zeit fast ausschließlich durch die Programmierung auf Assemblersprachen-Ebene möglich ist.

Aufgrund dieser engen Verflechtung von Algorithmenentwicklung und Programmierung soll im Folgenden der gesamte Bereich vom Problem zum Programm unter dem Oberbegriff Programmierung zusammengefaßt werden.

Die Problematik bei der Programmierung eines Pipelinerechners sowie der Versuch , bestimmte Probleme nach deren Eignung zur Lösung auf einem Pipelinerechner zu untersuchen, stellen die Hauptthemen dieses Kapitels.

Die Programmierung wird am Beispiel des AP 190 L vorgestellt. Bevor jedoch auf einige spezielle Problembereiche eingegangen wird, zunächst einige grundlegenden Gedanken zur Programmierung des AP.

4.2 Programmierbeispiel für den AP 190 L

Die Programmierung des AP erfolgt mittels der speziellen "Array Processor Assembly Language" (APAL). Diese Sprache kann als Mikroassembler angesehen werden, das heißt, die Befehle werden pro Takt direkt von der Hardware des AP ausgeführt.

Als einführendes Beispiel sei hier die Multiplikation eines Vektors der Länge N mit einem Skalarwert angegeben. Die Vorgehensweise bei der Lösung des Problems auf einem konventionellen Rechner kann durch die folgenden Punkte kurz skizziert werden:

1. Holen des Skalarwertes
2. Holen des ersten bzw. nächsten Vektorelementes
3. Multiplikation
4. Abspeichern des Ergebnisses
5. Bei Vektorende: Sprung nach 6.
   sonst: Sprung nach 2.
6. STOP

Bei der Lösung auf einem Pipelinerechner können die Aktivitäten der einzelnen Schleifendurchläufe aufgrund der Pipelinestruktur soweit wie möglich ineinandergeschachtelt werden, wodurch sich die Bearbeitungszeit zum Teil erheblich verringern läßt.

Für das folgende Beispiel kann davon ausgegangen werden, daß sich die zu verarbeitenden Daten im Datenspeicher und die entsprechenden Adressen in bestimmten S-PAD Registern befinden. Dies muß ansonsten vor Ausführungsbeginn der Subroutine sichergestellt sein.

Um das Zeitverhalten des AP deutlich werden zu lassen, dient die Reservierungstafel der wichtigsten Elemente des AP

(Bild 4.2). Alle Eintragungen in einer Spalte dieser Reservierungstafel werden in einem Takt ausgeführt. Die entsprechenden Zeilen haben folgende Bedeutungen:

MI: memory input register
Dieses Register wird zu Beginn eines Speicherschreibzyklus mit dem abzuspeichernden Wert geladen.

MA: memory address Register
Hierin ist entweder die Adresse eines zu lesenden Datenwortes oder die Adresse, auf die der Inhalt des MI abgelegt werden soll, jeweils zu Beginn eines Speicherzyklus enthalten.

MD: memory destination register
In diesem Register erscheint, drei Takte nach dem Starten eines Speicherlesezyklus durch setzen des MA, der Inhalt des gewünschten Speicherwortes.

TMA: table memory address register
Ein Setzen des TMA hat zur Folge, daß zwei Takte später der Inhalt des so angesprochenen Tabellenspeicherwortes im TM (table memory) Register verfügbar wird (Lesezyklus). Wird in den RAM-Teil des Tabellenspeichers geschrieben, so erhält TMA ebenfalls die entsprechende Adresse, der Wert jedoch wird über den Datenbus geschickt. Wie beim Schreibzyklus befindet sich dieser Wert zwei Zyklen später im TM Register.

TM: table memory register
Das TM Register dient als Ausgaberegister des Tabellenspeichers.

DPX/ data pad X/Y
DPY: Eintragungen in diese Zeilen umfassen zwei Komponenten:

1. Die Nummer des anzusprechenden Registers im entsprechenden Registerblock (relativ zum Pointer),

2. den abzuspeichernden Wert.

DB: data pad bus
Dieser 38 Bit breite Bus dient zum Transfer von Daten zwischen verschiedenen Komponenten.

FMUL: floating point multiplier
Hier wird das Starten einer Gleitpunktmultiplikation durch die Angabe der beiden Operanden vermerkt. Die beiden nachfolgenden Zeilen kennzeichnen das Schieben der Operanden durch die Stufen des Multiplizierers.

FM: floating point multiplier register
Ein Takt nach Schieben eines Operandenpaares in die letzte Pipelinestufe steht das Ergebnis in diesem Ausgaberegister des Multiplizierers zur Verfügung.

FADD: floating point adder
Die Eintragungen für den Addierer erfolgen wie beim Multiplizierer.

FA: floating point adder register
Ausgaberegister des Addierers, analog zu FM.

SPFN: S-Pad function
SPFN beinhaltet die Ausgabe der arithmetischen Einheit im S-Pad.

BR: branch
Sprungbedingung und Sprungadresse gemäß der in der ersten Zeile der Reservierungstafel angegebenen Nummerierung ist hier zu vermerken.

Bild 4.2 erlaubt einen Überblick über das Zeitverhalten des AP zur Lösung des Beispiels. Es sei vorausgesetzt, daß die Adresse des Skalarwertes, die Anfangsadressen von Eingabe- und Lösungsvektor und die Anzahl N der zu verarbeitenden Vektorelemente in speziellen S-Pad Registern zu Beginn der Abarbeitung verfügbar sind.

Die Spalten 1 bis 7 beschreiben die Aufsetzphase der Pipeline. Starten der Lesezyklen für Skalarwert (Nr.1) und die ersten beiden Vektorelemente (Nr.3 und 5), Zwischenspeichern des Skalarwertes S in DPX, Starten der 1. Multiplikation (Nr.6) sowie Adreßrechnungen und Schleifenzählen im S-Pad gehören zu dieser Aufsetzphase.

Die Spalten 8 bis 10 kennzeichnen die Schleife, die solange durchlaufen wird, bis die Sprungbedingung N = 0 erfüllt ist. Das bedeutet, pro Schleifendurchlauf wird ein Lesezyklus für ein neues Vektorelement gestartet und ein Ergebniswert abgespeichert. Aufgrund der Speicheroperationen wäre also eine Schleife, die nur 2 Takte benötigt, möglich. Es sind jedoch 3 S-Pad Operationen pro Schleife notwendig

1. Adresse für Lesezyklus bereitstellen,
2. Adresse für Speicherzyklus bereitstellen,
3. Zählen der verarbeiteten Elemente,

welche nur einzeln, das heißt pro Takt, ausgeführt werden können. Deshalb ist eine Verringerung der Taktzahl pro Schleifendurchlauf hier nicht möglich.

Von Spalte 11 bis 16 wird die Abarbeitung der beiden letzten Vektorelemente beendet.

Schleife

		1	2	3	4	5	6	7	8	9	10	11	12
DATA MEMORY	MI									E_{I-1}		E_{N-2}	
	MA	S		V_1		V_2			V_I	E_{I-2}		E_{N-2}	
			S		V_1		V_2			V_I			
				S		V_1		V_2			V_I		
	MD				S		V_1	V_2					
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX				S								
	DPY												
DATA BUS	DB												
MULTIPLIER	FMUL						$S=V_1$		$S=V_2$				
								$S=V_1$			$S=V_2$		$S=V_{N-1}$
									$S=V_1$				
	FM									E_1		E_{N-2}	
ADDER	FADD												
	FA												
SPAD	SPFN		N-1		N-2	V+1	E-1		V+I=1	E+I=1	N-I=1	E+N-2	
	BR								N=0: 11		8		

Bild 4.2: Reservierungstafel zur Lösung eines Skalar- Vektorproduktes

($E_i = S\ V_i$, $i = 3,\ldots,n$)

		13	14	15	16								
DATA MEMORY	MI		E_{N-1}		E_N								
	MA		E_{N-1}		E_N								
	MD	V_N											
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX												
	DPY												
DATA BUS	DB												
MULTIPLIER	FMUL	$S=V_N$											
			$S=V_N$										
		$S=V_{N-1}$		$S=V_N$									
	FM		E_{N-1}		E_N								
ADDER	FADD												
	FA												
SPAD	SPFN		E+N-1		E+N								
	BR												

Bild 4.2 : (Fortsetzung)

An diesem einfachen Beispiel zeigt sich bereits, daß die maximale Rechenleistung eines Pipelinerechners wohl nur sehr selten, bei speziellen Problemen erreicht werden kann.

Für den AP 190 L ist die maximale Rechenleistung nur zu erreichen, wenn pro Takt ein einziger Speicherzugriff erforderlich ist, da für jeden Speicherzugriff das MA Register benötigt wird. Im Programmierbeispiel waren jeweils ein Lese- und ein Schreibzyklus erforderlich.
Probleme mit nur einem Speicherzugriff sind in erster Linie von der Art, daß aus einer beliebigen Anzahl von Eingabewerten ein einziger Ergebniswert geliefert werden soll. Beispiel: die Summe der Elemente eines Vektors.

Neben diesen speziellen Problemen sind Vektor- und Matrixoperationen, trotz der obigen Einschränkung, bestens zur Lösung auf Pipelinerechnern geeignet, da aufgrund der Länge des zu verarbeitenden Datenstromes durch eine Überlappung der einzelnen Operationen eine wesentliche Verbesserung der Rechengeschwindigkeit gegenüber konventionellen Rechnern erzielt werden kann.

4.3 Effiziente Lösungen rekursiver Probleme

Probleme, welche Rekursionen beinhalten, scheinen zunächst einmal wesentlich ungeeigneter zur Lösung auf einem Pipelinerechner, als die bislang erwähnten Probleme. In den folgenden Abschnitten wird dies eingehend an bestimmten rekursiven Problemen untersucht.

4.3.1 Mathematische Grundlagen

Rekursive Probleme implizieren die Berechnung einer bestimmten Folge von Werten mittels einer Rekursionsformel. Die charakteristische Eigenschaft dieser Rekursionsformeln besteht darin, daß zur Berechnung eines Wertes ein oder mehrere zurückliegende Folgenglieder herangezogen werden.

Zwei einfache Beispiele zeigen diese Eigenschaft:

1. rekursive Definition der Summe:

$$\sum_{k=1}^{n} a_k = \sum_{k=1}^{n-1} a_k + a_n \quad , \quad \sum_{k=1}^{0} a_k = 0 \qquad (4.1)$$

oder mit $S_n = \sum_{k=1}^{n} a_k$

$$S_n = S_{n-1} + a_n \qquad (4.2)$$

2. rekursive Berechnung der Fibonaccizahlen:

$$F_n = F_{n-1} + F_{n-2} \quad , F_0=1, F_1=1 \qquad (4.3)$$

Diese beiden Rekursionsformeln unterscheiden sich unter anderem in einem wesentlichen Punkt: in Gleichung (4.2) wird nur auf den direkten Vorgänger, in Gleichung (4.3) auf das noch davor liegende Element der Folge zurückgegriffen

Definition: direkter Vorgänger

Das Element a_i einer Folge heißt direkter Vorgänger eines Elementes a_j, wenn der Abstand zwischen a_i und a_j gleich 1 ist.

Definition: Abstand

Der Abstand A zwischen zwei Folgenelementen a_i und a_j ist definiert als

$$A := j - i \quad (i \leq j)$$

Der maximale Abstand innerhalb der Folge zwischen dem zu berechnenden und den zur Berechnung zu verwendenden Elementen ergibt die Ordnung einer Rekursionsformel
Gleichung (4.2) hat demnach die Ordnung 1, Gleichung (4.3) besitzt die Ordnung 2 aufgrund des Abstandes zwischen F_n und F_{n-2}.
Allgemein ergibt sich also eine Rekursionsformel m-ter Ordnung zu

$$x_i = g(a_i, x_{i-1}, x_{i-2}, \ldots, x_{i-m}), \tag{4.4}$$

wobei die a_i lösungsunabhängige Parameter und g eine beliebige (m+1)-stellige Funktion darstellen. Eine Teilmenge dieser Rekursionsformeln bilden die Rekursionsformeln 1. und 2. Ordnung, die auch als Differenzengleichungen bezeichnet werden.
Die Gleichungen (4.1) und (4.2) sind Spezialfälle der allgemeinen linearen Rekursionsformel 1.Ordnung

$$x_i = a_i \cdot x_{i-1} + f_i \tag{4.5}$$

da sie für $a_i = 1$ aus (4.5) hervorgehen.
Die allgemeine lineare Rekursionsformel 2.Ordnung ist von der Form

$$x_i = a_i \cdot x_{i-1} + b_i \cdot x_{i-2} + f_i \tag{4.6}$$

Für $f_i = 0$ entsteht aus (4.5) eine homogene Rekursionsformel 1.Ordnung

$$x_i = a_i \cdot x_{i-1} \tag{4.7}$$

und aus (4.6) eine homogene Rekursionsformel 2.Ordnung

$$x_i = a_i \cdot x_{i-1} + b_i \cdot x_{i-2} \tag{4.8}$$

Die Gleichung (4.3) stellt einen Spezialfall der Gleichung (4.8) dar mit $a_i = b_i = 1$.

Diese Differenzengleichungen, wie sie im Folgenden bezeichnet werden sollen, finden in erster Linie breite Anwendung im Bereich der Numerischen Mathematik, treten dort in der Regel als diskrete Approximationen für Differentialgleichungen auf /Ber 79/ und sind daher auf Gebieten der Physik, der Mechanik und der Elektrotechnik von besonderem Interesse. Aus diesem Grunde soll die Problematik der Lösung solcher Gleichungen, die zum Teil sehr rechenintensiv sein können, unter dem Gesichtspunkt der Effizienz auf Pipelinerechnern in den folgenden Abschnitten untersucht werden.

4.3.2 Rekursion und Pipelining

Die beiden Vorgehensweisen bei der Rekursion und beim Pipelining scheinen zunächst einmal nicht miteinander vereinbar zu sein, denn bei der Rekursion wird zur Berechnung eines Wertes einer oder mehrere seiner Vorgänger herangezogen, während der eigentliche Sinn des Pipelining darin besteht, daß die Berechnung eines Wertes schon beginnen kann, obwohl sich eine bestimmte Anzahl von Vorgängern noch in der Berechnungsphase befinden.

Das Problem wird deutlich, wenn die Lösung der Funktion g aus (4.4) durch eine Pipeline mit d_g Stufen (=d_g Zeiteinheiten) realisiert wird /Kog 73/. Dann müssem die einmal berechneten x-Werte wieder rückgekoppelt werden, um zur Berechnung ihrer Nachfolger zur Verfügung zu stehen (Bild 4.3). Dies geschieht hier unter Verwendung von zwischengeschalteten Puffern.
Wie in Bild 4.4 zu sehen ist, steht nur nach jeweils d_g Zeiteinheiten ein neuer x-Wert für weitere Rechnungen bereit, da die (i-1). Berechnung vollkommen abgeschlossen sein muß, bevor die Berechnung für x_i beginnen kann.

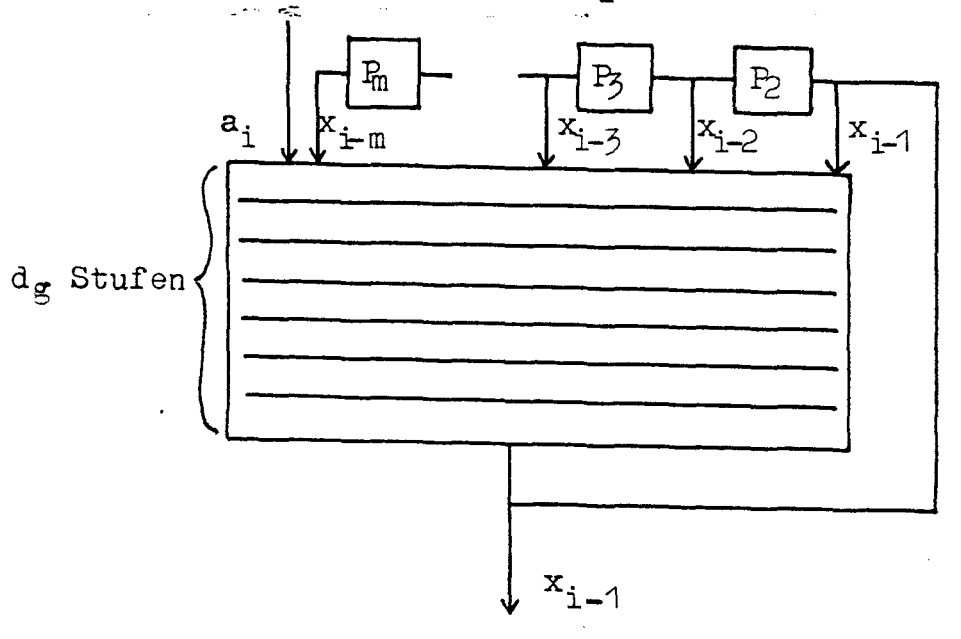


Bild 4.3: Diskrete Berechnung von g durch eine Pipeline

Dies bietet jedoch keine Verbesserung gegenüber der Lösung auf einem konventionellen Rechner (ohne Pipeline), das heißt, rekursive Probleme müssen entsprechend der jeweils vorliegenden Pipelinestruktur umgeformt werden, um dann eine bessere Auslastung der Pipeline und somit einen Speedup größer als 1 zu erreichen.

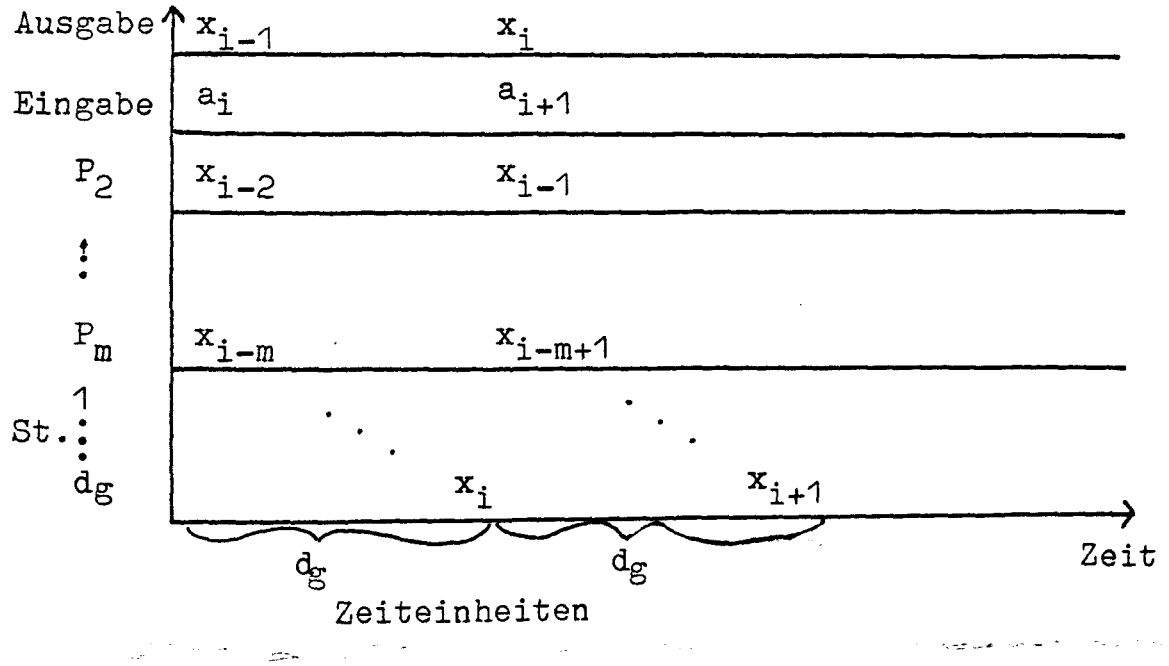


Bild 4.4: Zeitverhalten bei der Berechnung von g

Das Ziel der folgenden Untersuchungen wird also sein, die Ergebnisrate - hier 1/d - so zu verbessern, daß sie möglichst nahe bei 1 liegt.
Dies soll zunächst am konkreten Beispiel der Summation von n Zahlen $a_1, a_2, \ldots a_n$ (Gleichung (4.1) bzw. (4.2)) versucht werden.

$$S_n = S_{n-1} + a_n \quad , S_o=0, \; i=1,2,3,\ldots \qquad (4.9)$$

Die Hinzunahme der Anfangsbedingung $S_o = 0$ ermöglicht jetzt die Berechnung der S_i. Für die Lösung des Problems stehe ein 3-stufiger Gleitpunktaddierer zur Verfügung.
Die Reservierungstafel in Bild 4.5 läßt erkennen, daß die Ergebnisrate 1/3 beträgt. Der gesamte Zeitbedarf der Lösung beläuft sich auf $T_n = 3\,n$ Zeiteinheiten.

Unter der Berücksichtigung der vorliegenden Pipeline läßt sich das Problem, allerdings unter Verlust der Zwischenergebnisse S_i ($1 \leq i \leq n-1$), umformen, indem unabhängig voneinander drei Teilsummen gebildet werden. Diese Teilsummen werden dann zum Ende der Berechnung nur noch aufsummiert.

$$S_n^* = S_1^* + S_2^* + S_3^* \qquad \text{mit } S_i^* = \sum_{k=i+m\,3}^{1} a_k \,, \qquad (4.10)$$

$$m = 0,1,2,\ldots \,,$$

$$i \quad 1,2,3 \,,\; 1 \leq n$$

Bild 4.6 zeigt die Reservierungstafel für das umgeformte Problem. Die Ergebnisrate

$$R = \frac{n+2}{n+7} \qquad (\lim_{n\to\infty} R = 1) \qquad (4.11)$$

und der Speedup

$$S = \frac{3n}{n+7} \qquad (S>1 \text{ für } n \geq 5) \qquad (4.12)$$

machen die Verbesserung der Pipelineausnutzung deutlich.

Eine solche Umformung läßt sich jedoch nur bei rekursiven Problemen mit ganz besonderen Eigenschaften durchführen:

a) Assoziativität der zu benutzenden Teilfunktion (hier Addition)

b) Möglichkeit der unabhängigen Teilfunktionsberechnungen, das heißt, rückführen einer rekursiven Funktion auf unabhängige Teilrekursionen.

Allgemein gilt also:
gegeben: - Rekursionsproblem mit obigen Charakeristika und
- k-stufige Pipeline zur Berechnung der gewünschten Funktion f.
Dann sind k unabhängige Teilfunktionsberechnungen und k-1 Abschlußberechnungen durchzuführen, die ihrerseits für k-1 ≥ 4 wieder in unabhängige Teilberechnungen aufgeteilt werden können u.s.w.

So ergibt sich bei Verknüpfung von n Elementen durch eine 2-stellige Funktion eine obere Schranke für die Rechenzeit von

$$T_s = t_1 + t_2 + t_3 = k-1 + n + k(k-1) \qquad (4.13)$$

mit t_1 = Aufsetzzeit der Pipeline,
t_2 = Zeit, in der pro Takt ein Ergebnis erzeugt wird,
t_3 = Zeit zur Schlußberechnung.

Der Speedup gegenüber der normale Ausführung lautet dann

$$S = \frac{T_1}{T_s} = \frac{d \cdot n}{d^2-1+n} \quad \text{mit } S \;\; 1 \text{ für } n \;\; d+1 \qquad (4.14)$$

Da aber nur sehr wenige rekursive Probleme die oben genannten Eigenschaften besitzen, ein weiteres Beispiel ist die Berechnung der Fakultät

$$n! = (n-1)!\; n \,, \quad n \quad , \; 0! = 1 \qquad (4.15)$$

müssen die Überlegungen jetzt auf allgemeinere Rekursionsformeln erweitert werden.

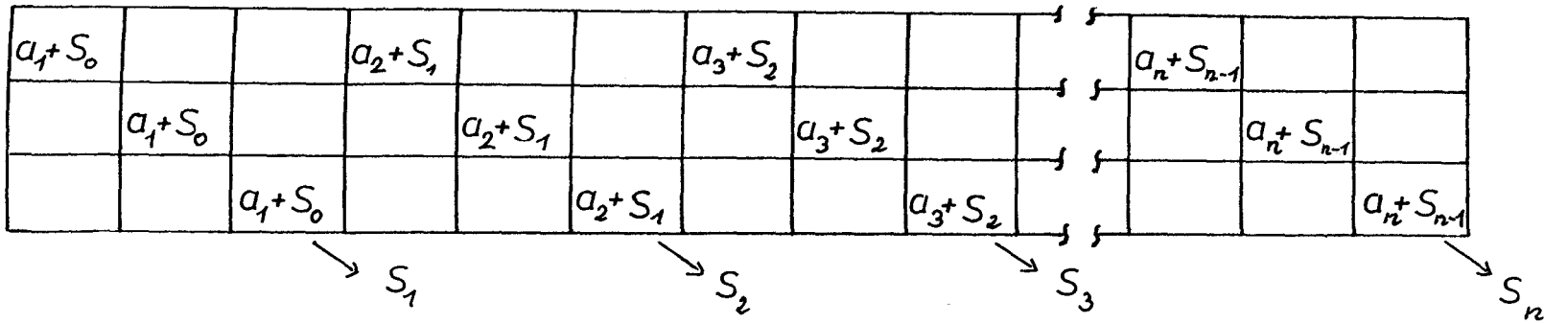


Bild 4.5: Reservierungstafel für einen 3-stufigen Addierer zur Berechnung von $S_n = S_{n-1} + a_n$ (4.9)

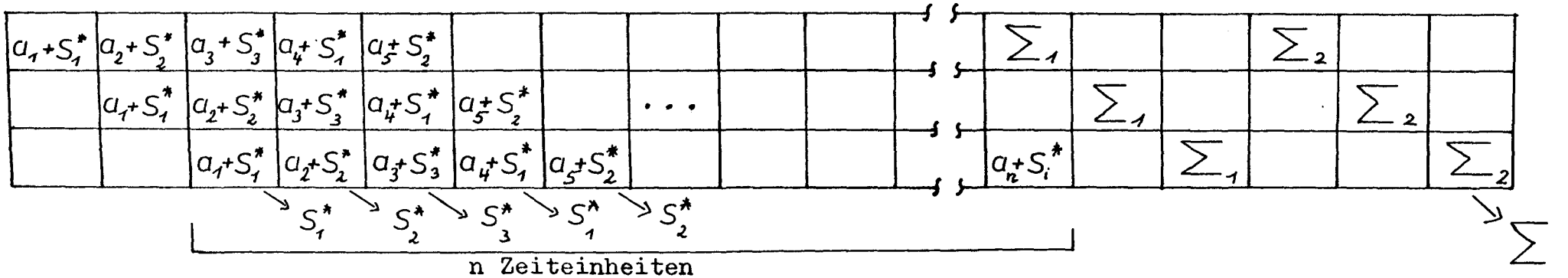


Bild 4.6: Reservierungstafel für einen 3-stufigen Addierer zur Berechnung von (4.1o)

Allgemein läßt sich also für lineare Rekursionsformeln 1. Ordnung, für die (4.9) und (4.15) zwei spezielle Beispiele sind, die vorstehende Problemumformung nicht durchführen.
Um zum Beispiel für die homogene Differenzengleichung 1. Ordnung (4.7)

$$x_i = a_i \cdot x_{i-1} \tag{4.16}$$

eine Verbesserung der Ergebnisrate gegenüber der konventionellen Lösung bzw. einen gewissen Speedup zu erreichen, bleibt hier nur die Möglichkeit, bei der Berechnung von x_i nicht auf den direkten Vorgänger, der sich infolge des Pipelinings noch in der Bearbeitungsphase befindet, sondern auf einen weiter davor liegenden, bereits berechneten Wert zurückzugreifen.
Für eine k-stufige Pipeline bietet sich also an, die Gleichung (4.16) auf

$$x_i = a_i \cdot a_{i-1} \cdot a_{i-2} \cdot \ldots \cdot a_{i-k+1} \cdot x_{i-k} \tag{4.17}$$

zurückzuführen. Diese Art der Problemumformung soll im Folgenden mit Rekursionsreduktion bezeichnet werden. Um an einem konkreten Fall die Problematik der Rekursionsreduktion zu zeigen, reicht bereits k = 2 aus und (4.17) ergibt sich als

$$x_i = a_i \cdot a_{i-1} \cdot x_{i-2} \tag{4.18}$$

Eine Egebnisrate von 1/2 - bei konventioneller Lösung erreichbar - ließe sich jedoch auch jetzt noch nicht verbessern, weil zwei Multiplikationen hintereinander erforderlich wären, die sich nicht überlappen lassen. Da jetzt insgesamt eine Zeitverzögerung von 4 Zeiteinheiten für eine Berechnung entsteht, müssen zwei weitere Schritte der Rekursionsreduktion vorgenommen werden:

$$\begin{aligned} x_i &= a_i \cdot a_{i-1} \cdot a_{i-2} \cdot x_{i-3} \\ x_i &= \underline{a_i \cdot a_{i-1}} \cdot a_{i-2} \cdot a_{i-3} \cdot x_{i-4} \end{aligned} \tag{4.19}$$

was aber eine weitere Erhöhung der Anzahl der Multiplikationen zur Folge hat. Dies kann jedoch umgangen werden unter Berücksichtigung bereits im vorhergehenden Schritt ermittelter Teilprodukte

$$\begin{aligned} x_{i+1} &= a_{i+1} \cdot \underline{a_i \cdot a_{i-1}} \cdot a_{i-2} \cdot x_{i-3} \\ x_{i+2} &= a_{i+2} \cdot a_{i+1} \cdot \underline{a_i \cdot a_{i-1}} \cdot x_{i-2} \end{aligned}$$

Da die Berechnung optimal in der Form einer Binärbaumstruktur (Bild 4.7) durchgeführt wird, ergibt sich für n Elemente eine minimale Anzahl von $\lceil \log_2 n \rceil$ hintereinander auszuführenden Operationen, was einer Berechnung in $\lceil \log_2 n \rceil$ Ebenen entspricht.

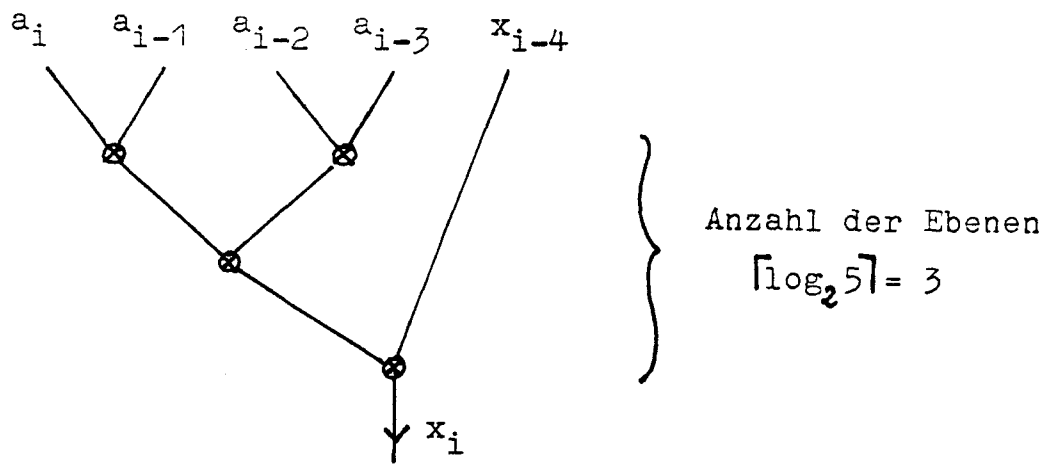


Bild 4.7: Baumstruktur für Multiplikationen von n = 5 Elementen

Aus Bild 4.7 ist bereits ein wichtiges Ergebnis ableitbar: ohne eine gewisse Anzahl von parallel arbeitenden Funktionseinheiten ist das angestrebte Ziel, 1 Ergebnis pro Zeittakt zu erhalten, nicht zu verwirklichen.
Die Gesamtzahl m der benötigten Funktionseinheiten läßt sich aus den oben erwähnten Gründen durch Zwischenschalten von Puffern zwar noch etwas verringern - zum Beispiel bei 2-stufigen Funktionseinheiten und 7 zu verknüpfenden Elementen von m = 6 auf m = 4 /Kog 73/-, auf die Anzahl der Bearbeitungsebenen hat dies jedoch keinen Einfluß.

Ein Maß für die Anzahl der erforderlichen Reduktionsschritte ergibt sich direkt aus den zur Berechnung eines x_i benötigten Zeiteinheiten (= Anzahl der Bearbeitungsebenen * Anzahl der Pipelinestufen pro Ebene).
Für k-stufige Multiplikationspipelines und n - 1 durchzuführende Multiplikationen (bei n Elementen) muß die Ungleichung

$$n - 1 \geq k \lceil \log_2 n \rceil \qquad (4.20)$$

erfüllt sein, um eine maximale Ergebnisrate zu erreichen.
n - 1 Multiplikationen implizieren aber auch
(n-1) - 1 = n - 2 Rekursionsreduktionen für den Fall der homogenen Differenzengleichung 1. Ordnung.

<u>Beispiel</u>: k=2

$n-1 \geq 2 \cdot \lceil \log_2 n \rceil$ $\qquad$ $n \geq 7$

5 Reduktionen

$x_i = a_i \cdot x_{i-1}$

$x_i = a_i \cdot a_{i-1} \cdot x_{i-2}$ $\qquad$ 1.Reduktion

$\vdots$ $\qquad$ $\vdots$

$x_i = a_i \cdot a_{i-1} \cdot a_{i-2} \cdot a_{i-3} \cdot a_{i-4} \cdot a_{i-5} \cdot x_{i-6}$

Die Anzahl m der insgesamt benötigten Multiplizierer mit

$$\lceil \log_2 n \rceil \leq m \leq n - 1 \qquad (4.21)$$

richtet sich jeweils nach der Möglichkeit, Puffer zur Aufbewahrung von Teilergebnissen zwischenzuschalten. Diese Puffer stellen die Zwischenergebnisse für nachfolgende Berechnungen bereit.
Ganz analog gestaltet sich auch die Vorgehensweise bei Differenzengleichungen 2. Ordnung (4.8). Als einfaches Beispiel ist die Berechnung der Fibonaccizahlen anzusehen

$$F_n = F_{n-1} + F_{n-2} \qquad (4.22)$$

Gegeben sei ein Pipelinerechner mit k-stufigem Addierer. Das macht Rekursionsreduktionen erforderlich, bis auf die Werte F_{n-k} und F_{n-k-1} zurückgegriffen werden kann. Mit

$$F_{n-1} = F_{n-2} + F_{n-3}$$

eingesetzt in (4.22) folgt

$$F_n = 2 \cdot F_{n-2} + F_{n-3} \qquad \text{1.Red.}$$

und durch weitere Einsetzungen erhält man

$$\begin{aligned} F_n &= 3 \cdot F_{n-3} + 2 \cdot F_{n-4} && \text{2.Red.} \\ &= 5 \cdot F_{n-4} + 3 \cdot F_{n-5} && \text{3.Red.} \\ &= 8 \cdot F_{n-5} + 5 \cdot F_{n-6} && \text{4.Red.} \\ &\vdots \end{aligned}$$

Die Koeffizienten bilden, jeweils um 1 versetzt, wiederum die Folge der Fibonaccizahlen, so daß allgemein für die (k-1)-te Reduktionsstufe gilt:

$$F_n = F_k \cdot F_{n-k} + F_{k-1} \cdot F_{n-k-1} \qquad (4.23)$$

Die jetzt zusätzlich erforderlichen Multiplikationen seien auf l-stufigen Multiplizierern parallel ausführbar. Da diese aber nicht mit der Addition überlappt werden können, sind weitere l Reduktionsschritte notwendig, um eine maximale Ergebnisrate erreichen zu können. Insgesamt wird dann also die (k+l-1)-te Reduktionsstufe erreicht, und die Berechnung erfolgt nach der Gleichung

$$F_n = F_{k+l} \cdot F_{n-k-l} + F_{k+l-1} \cdot F_{n-k-l-1} \qquad (4.24)$$

Die Koeffizienten sind unabhängig von n und können daher einmal vorab berechnet werden. So ergibt sich also mit fortschreitender Rekursionsreduktion keine Erhöhung der Anzahl der benötigten Operationen und für (4.24) die optimale Ergebnisrate von 1 Ergebnis pro Zeittakt, vorausgesetzt, zwei Multiplizierer können parallel arbeiten.

Wie in diesem speziellen Beispiel der Fibonaccizahlen sieht

auch die Problemlösung für die ganze Klasse der homogenen Differenzengleichungen 2. Ordnung (4.8) mit konstanten Koeffizienten aus. Die Gleichungen für die Rekursionsstufen lassen sich ebenfalls durch wiederholtes Einsetzen ermitteln:

$$x_i = ax_{i-1} + bx_{i-2} \tag{4.25}$$

$$= (a^2+b)x_{i-2} + abx_{i-3} \tag{4.25a}$$

$$= (a^3+2ab)x_{i-3} + (a^2b+b^2)x_{i-4} \tag{4.25b}$$

$$= (a^4+3a^2b+b^2)x_{i-4} + (a^3b+2ab^2)x_{i-5} \tag{4.25c}$$

$$= (a^5+4a^3b+3ab^2)x_{i-5} + (a^4b+3a^2b^2+b^3)x_{i-6} \tag{4.25d}$$

$$\vdots$$

Da a und b konstant sind, können die Koeffizienten der gewünschten Reduktionsstufen wiederum einmal berechnet und dann als konstante Faktoren jeweils mit den x-Werten multipliziert werden. Das heißt, nach einer gewissen Vorbereitungszeit zur Berechnung der Koeffizienten und der benötigten Anfangswerte ist auch hier die maximale Ergebnisrate zu erreichen.

Allgemein gilt:

Gegeben sei ein Pipelinerechner mit k-stufigen Addierern und l-stufigen Multiplizierern. Dann muß zur effizienten Lösung homogener Differenzengleichungen 2. Ordnung mit konstanten Koeffizienten eine (k+l-1)-malige Rekursionsreduktion durchgeführt werden:

$$x_i = A_1x_{i-1} + B_1x_{i-2}$$
$$= A_2x_{i-2} + B_2x_{i-3} \qquad \text{1.Reduktion}$$
$$\vdots \qquad\qquad \vdots$$
$$= A_{k+l}x_{i-k-l} + B_{k+l}x_{i-k-l-1}$$

mit: $A_j = A_{j-1}\,a + B_{j-1}$,
$B_j = A_{j-1}\,b$,
$A_1 = a$,
$B_1 = b$ und $j \in \mathbb{N}$

Beim Übergang von homogenen zu <u>inhomogenen Differenzengleichungen</u> 2. Ordnung mit konstanten Koeffizienten gestalten sich die Gleichungen für die einzelnen Reduktionsstufen schon weitaus komplexer:

$$0.\ x_i = ax_{i-1} + bx_{i-2} + f_i$$

$$I.\ x_i = (a^2+b)x_{i-2} + abx_{i-3} + af_{i-1} + f_i$$

$$II.\ x_i = (a^3+2ab)x_{i-3} + (a^2b+b^2)x_{i-4} + af_{i-2} + (a^2+b)f_{i-1} + f_i$$

$$\vdots$$

$$M.\ x_i = A_M x_{i-M-1} + B_M x_{i-M-2} + A_{M-1} f_{i-1} + A_{M-2} f_{i-2} \ \ldots$$
$$\ldots + A_0 f_{i-M} + f_i$$

mit: A_j, B_j wie oben definiert und
M = Anzahl der Reduktionsstufen

Auch hier können wieder die Koeffizienten, die sich aus den Konstanten a und b ergeben, einmal vorab berechnet werden, allerdings erhöht sich jetzt die Anzahl der benötigten Operationen pro zusätzlichem Reduktionsschritt um eine Addition und eine Multiplikation.
Unter der Annahme, daß die M+1 erforderlichen Multiplikationen parallel auf k-stufigen Multiplizierern ausgeführt werden, ergibt sich ein gesamter Zeitaufwand von

$$t = k + 1 \left\lceil \log_2(M+3) \right\rceil \qquad (4.26)$$

und für

$$M \geq k + 1 \left\lceil \log_2(M+3) \right\rceil \qquad (4.27)$$

ist die maximal mögliche Ergebnisrate auch in diesem Fall erreichbar.
Auch hier ist wieder ein hoher Grad an Parallelität erforderlich, um höchstmögliche Effizienz zu erzielen. Eine weitere Komplizierung des Problems tritt auf, wenn statt konstanten Koeffizienten variable Koeffizienten $\alpha = (a_1, a_2, \ldots)$ und $\beta = (b_1, b_2, \ldots)$ vorliegen. Dann nämlich sind die Koeffizienten für die entsprechende Reduktionsstufe für jeden x-Wert neu zu berechnen, und das eigentlich angestrebte Ziel, hohe Ergebnisraten, scheint immer weiter in die Ferne zu rücken, wenn nicht eine große Anzahl Funktionseinheiten bei der Lösung solcher Probleme parallel arbeiten kann.

Die aus den bisherigen Beispielen gewonnenen Kenntnisse ermöglichen jetzt die Angabe einer Vorgehensweise zur Umformung eines rekursiven Problems, damit dieses Problem auf einem speziellen Pipelinerechner effizient lösbar wird. Diese Vorgehensweise wird in Form eines Algorithmus beschrieben:

0. i := 0 ; s := 0 (s = gesamte Zeitverzögerung, $i \in \mathbb{N}$)

1. Aufstellen der Gleichung für die i-te Reduktionsstufe

2. Angabe der sequentiell (problembezogen) auszuführenden Operationen

3. s := s + Summe der aus 2. resultierenden Pipelinestufen; Angabe der parallel arbeitenden Pipelines

4. Ist i < s, dann i := i + 1 und gehe zu 1.
   sonst gehe zu 5.

5. STOP

Sind alle p parallel angeforderten Funktionseinheiten verfügbar, dann ist die Ergebnisrate R = 1, sind nur p - q Pipelines verfügbar, so verschlechtert sich die Ergebnisrate auf

$$R = \frac{1}{1+q} \qquad (4.28)$$

Eine wichtige Frage, deren Beantwortung für die Durchführbarkeit der gezeigten Lösungswege unerläßlich ist, ist die nach der Speicher- und Busbeschaffenheit. Schon bei den einfachen Problemen ist eine Vielzahl parallel arbeitender Funktionseinheiten erforderlich, um eine möglichst hohe Ergebnisrate zu erreichen. Das heißt aber wiederum, mehrere Funktionseinheiten müssen im gleichen Takt mit unterschiedlichen Eingabedaten versorgt werden.

Dies ist jedoch für jedes Problem in Abhängigkeit von der vorliegenden Rechnerstruktur separat zu klären, da zum Beispiel die Anzahl der Pipelinestufen die Anzahl der Rekursionsreduktionen und damit auch den Grad der geforderten Parallelität bestimmt. Einfach ausgedrückt: je größer die Anzahl der Stufen in einer arithmetischen Pipeline ist, je höher sind auch die Parallelitätsanforderungen.

Die erörterten theoretischen Lösungsansätze werden in den folgenden Abschnitten auf einen realen Pipelinerechner angewandt und am Beispiel des FPS AP190L - auch anhand von Testprogrammen- nachvollzogen.

4.3.3 Lösungen auf dem FPS AP190L

Als ein wichtiges Merkmal der heute verfügbaren Pipelinerechnersysteme ist festzuhalten, daß für Gleitpunktoperationen wie Addition, Multiplikation, Division meist nur jeweils eine Funktionseinheit zur Verfügung steht. Das hat zur Folge, daß eine Ergebnisrate R = 1 in den seltensten Fällen erreichbar ist. Allgemein ist die erzielbare Ergebnisrate (28) mit p-q = 1 $\Rightarrow$ q = p-1 anzugeben als

$$R = \frac{1}{p}\ , \qquad (4.29)$$

- Homogene Differenzengleichungen 2.Ordnung mit konstanten Koeffizienten (4.25)

$$x_i = a \cdot x_{i-1} + b \cdot x_{i-2}$$

Zwei parallel arbeitende Multiplizierer sind für eine Ergebnisrate R = 1 erforderlich, vorhanden ist jedoch nur eine Funktionseinheit (p-q=1), und daraus folgt:

$$R = \frac{1}{1+q} = \frac{1}{1+p-1} = \frac{1}{2} \qquad (4.3o)$$

ist die im besten Fall erreichbare Ergebnisrate.
Die direkte Lösung - ohne Rekursionsreduktionen - von (4.25) liefert eine Ergebnisrate von R = 1/5 (Bild 4.8). Da aber aufgrund der Doppelbenutzung der Multiplikationseinheit bestenfalls nur in jedem zweiten Takt ein neues Ergebnis verfügbar wird, sind auch nur (k+l-1)/2 Rekursionsreduktionen durchzuführen, um die für den vorliegenden Rechner beste Ergebnisrate zu erreichen. Mit dem 2-stufigen Addierer (k = 2) und dem 3-stufigen Multiplizierer (l = 3) erhält man für den AP nach (2+3-1)/2 = 2 Reduktionsschritten (4.25b)

$$x_i = (a^3 + 2ab)\, x_{i-3} + (a^2 b + b^2)\, x_{i-4}$$

Daß die Ergebnisrate (4.30), die sich aus den Parallelitäts-anforderungen des Problems ergibt, tatsächlich erreichbar ist, zeigen die folgenden Überlegungen.

Das erste Ziel bei der Lösung des Problems besteht darin, die Aufsetzzeit möglichst gering zu halten. Dazu läßt sich die Anzahl der nötigen Rechenoperationen zur einmaligen Berechnung der Koeffizienten minimieren, indem durch Ausklammern die Exponenten reduziert werden:

$a^3 + 2ab$	4 Multiplikationen + 1 Addition
$a(a^2 + 2b)$	3 Multiplikationen + 1 Addition
$a^2 b + b^2$	3 Multiplikationen + 1 Addition
$b(a^2 + b)$	2 Multiplikationen + 1 Addition

a^2 kommt in beiden Koeffizienten vor, daher ergibt sich eine Gesamtzahl der benötigten Operationen von 4 Multiplikationen und 2 Additionen. Unter der Berücksichtigung, daß 2*b in kürzerer Zeit durch b+b berechnet werden kann, kommt man letztlich auf jeweils 3 Multiplikationen und Additionen.
Die Berechnung der beiden Koeffizienten zeigt Bild 4.9.

Damit ist die Vorbereitungsphase jedoch noch nicht abgeschlossen, denn (25b) berechnet erst x_i-Werte für $i \geq 4$ bei Vorgabe von x_0 und x_1. So sind also x_2 und x_3 noch zu berechnen, bevor ein reibungsloser Ablauf der Berechnung gewährleistet ist. Dazu werden x_2 und x_3 auf folgende Weise berechnet:

$$x_2 = a \cdot x_1 + b \cdot x_0$$

$$x_3 = (a^2+b) \cdot x_1 + abx_0$$

Die Frei-Zeiten der Pipelines während der Berechnung der Koeffizienten werden nach Möglichkeit für die Berechnung der beiden x-Werte ausgenutzt. Bild 4.10 zeigt, daß die erwartete Ergebnisrate R = 1/2 nach einer Aufsetzzeit von 20 Zeiteinheiten erreicht werden kann.

Das Bereitstellen der gewünschten Eingabedaten bereitet keine Schwierigkeiten, da nur die Anfangswerte x_0 und x_1 sowie die Ausgangskoeffizienten a und b einmal aus dem Speicher zu holen sind. Alle anderen Zwischenergebnisse können in den schnellen Data Pads zwischengespeichert werden.

Insgesamt ist also ein Speedup von S = 2.5 gegenüber der direkten Lösung ohne Rekursionsreduktionen zu erreichen.

Das Assembler-Programm zur effizienten Lösung homogener Differenzengleichungen 2. Ordnung mit konstanten Koeffizienten befindet sich im Anhang sowie eine Kurzbeschreibung der wichtigsten Assembler-Befehle befindet sich im Anhang.

		1	2	3	4	5	6	7	8	9	10	11	12
DATA MEMORY	MI												X_2
	MA	A	B	X_0	X_1								X_2
			A	B	X_0	X_1							
				A	B	X_0	X_1						
	MD				A	B	X_0	X_1					
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX				0<A			1<X_1					1<X_2
	DPY					0<B							
DATA BUS	DB												
MULTIPLIER	FMUL						B=X_0	A=X_1				B=X_1	A=X_2
								B=X_0	A=X_1				B=X_1
									B=X_0	A=X_1			
	FM									B=X_0	A=X_1		
ADDER	FADD										S_1		
												S_1	
	FA												X_2
SPAD	SPFN	A	B	X_0	X_1								
	BR												

Bild 4.8: Reservierungstafel für die direkte Lösung von $x_i = ax_{i-1} + bx_{i-2}$

		13	14	15	16	17	18	19	20	21	22		
DATA MEMORY	MI					X_3					X_4		
	MA					X_3					X_4		
	MD												
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX					$1<X_3$					$1<X_4$		
	DPY												
DATA BUS	DB												
MULTIPLIER	FMUL				$B=X_2$	$A=X_3$							
		$A=X_2$				$B=X_2$	$A=X_3$						
		$B=X_1$	$A=X_2$				$B=X_2$	$A=X_3$					
	FM		$B=X_1$	$A=X_2$				$B=X_2$	$A=X_3$				
ADDER	FADD			S_2					S_3				
					S_2					S_3			
	FA					X_3					X_4		
SPAD	SPFN												
	BR												

Bild 4.8: (Fortsetzung)

		1	4	5	6	7	8	9	10	11	12	13
DATA MEMORY	MI											
	MA	A										
			B									
				B								
	MD		A		B							
TABLE MEMORY	TMA											
	TM											
DATA PADS	DPX				0<B			1<K_2	2<K_1			
	DPY											
DATA BUS	DB											
MULTIPLIER	FMUL		A=A					K_2	K_1			
				A=A					K_2	K_1		
					A=A					K_2	K_1	
	FM					A=A					K_2	K_1
ADDER	FADD				B+B	A=2+B	A=2+2B					
						B+B	A=2+B	A=2+2B				
	FA						2B	A=2+B	A=2+2B			
SPAD	SPFN											
	BR											

Bild 4.9: Reservierungstafel für die Berechnung der Koeffizienten aus 4.25b mit $K_1 = a^3 + 2ab$

$K_2 = a^2b + b^2$

		1	2	3	4	5	6	7	8	9	10	11	12
DATA MEMORY	MI												
	MA	A	B	X_1	X_0								
			A	B	X_1	X_0							
				A	B	X_1	X_0						
	MD				A	B	X_1	X_0					
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX				0<A			1<X_0		2<A≡A	-1<AX_1	-2<A≡A+B	
	DPY					0<B	1<X_1						2<A≡A+2B
DATA BUS	DB												
MULTIPLIER	FMUL						A≡A	A≡X_1	B≡X_0	A≡B		(A≡A+B)X_1	AB≡X_0
								A≡A	A≡X_1	B≡X_0	A≡B		(A≡A+B)X_1
									A≡A	A≡X_1	B≡X_0	A≡B	
	FM									A≡≡2	A≡X_1	B≡X_0	A≡B
ADDER	FADD								B+B	A≡A+B	A≡A+2B	S_2	
										B+B	A≡A+B	A≡A+2B	S_2
	FA										2B	A≡A+B	A≡A+2B
SPAD	SPFN	A	B	X_1	X_0								
	BR												

Bild 4.10: Reservierungstafel zur Berechnung homogener Differenzengleichungen 2. Ordnung mit konstanten Koeffizienten

		13	14	15	16	17	18	19	20	21	22	23	24
DATA MEMORY	MI	X_2							X_3		X_4		
	MA	X_2							X_3		X_4		
	MD												
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX		0<(FM)		$0<K_1$								
	DPY	$2<X_2$				$3<K_2$		$-1<K_1X_1$	$0<X_3$	$-1<K_1X_2$	$2<X_4$	$-1<K_1X_3$	
DATA BUS	DB												
MULTIPLIER	FMUL	A(A:=A+2B)	B(A:=A+B)		K_1X_1	K_2X_0	K_1X_2	K_2X_1	K_1X_3	K_2X_2	K_1X_4	K_2X_3	
		AB:=X_0	A(..)	B(..)		K_1X_1	K_2X_0	K_1X_2	K_2X_1	K_1X_3	K_2X_2	K_1X_4	
		(A:=A+B)X_1	AB:=X_0	A(..)	B(..)		K_1X_1	K_2X_0	K_1X_2	K_2X_1	K_1X_3	K_2X_2	
	FM		(A:=A+B)X_1	AB:=X_0	K_1	K_2		K_1X_1	K_2X_0	K_1X_2	K_2X_1	K_1X_3	
ADDER	FADD			S_3					S_4		S_5		
								S_3		S_4		S_5	
	FA	X_2							X_3		X_4		X_5
SPAD	SPFN	X_2							X_3		X_4		X_5
	BR												

Bild 4.10: (Fortsetzung)

- Inhomogene Differenzengleichungen
  2.Ordnung mit konstanten Koeffizienten

$$x_i = ax_{i-1} + bx_{i-2} + f_i \qquad (4.31)$$

Die Ergebnisrate lautet auch hier, wie im homogenen Fall, R = 1/2 (Bild 4.11), da die zusätzlich benötigte Addition während der zweiten Multiplikation ausgeführt werden kann. Führt man eine Rekursionsreduktion durch. so ergibt das

$$\begin{aligned} x_i &= ax_{i-1} + bx_{i-2} + f_i \\ &= a(ax_{i-2}+bx_{i-3}+f_{i-1}) + bx_{i-2} + f_i \\ &= (a^2+b)x_{i-2} + abx_{i-3} + af_{i-1} + f_i \end{aligned} \qquad (4.32)$$

Wie bereits gezeigt, werden in diesem Fall pro Reduktionsschritt eine Multiplikation und eine Addition zusätzlich erforderlich. Nach kurzer Aufsetzzeit zur Berechnung der Koeffizienten a +b und ab sowie des Wertes x stellt sich eine Ergebnisrate von R = 1/3 ein (Bild 4.12).

Eine Verbesserung dieser Ergebnisrate ist für den AP nicht mehr zu erreichen, da bereits hier eine fast volle Ausnutzung der arithmetischen Pipelines festzustellen ist. Der maximal erreichbare Speedup beträgt also S = 5/3.

- Weitere Differenzengleichungen

Für die Lösungen von Differenzengleichungen m-ter Ordnung mit konstanten Koeffizienten ist im homogenen Fall

$$x_i = a_1x_{i-1} + a_2x_{i-2} + \ldots + a_mx_{i-m} \qquad (4.33)$$

eine maximale Ergebnisrate von R = 1/m durch entsprechende Rekursionsreduktionen möglich, da dies durch die m unabhängig hintereinander auszuführenden Multiplikationen bestimmt wird.

Inhomogene Differenzengleichungen m-ter Ordnung lassen sich durch die Problemumformung mittels Rekursionsreduktion kaum effizienter berechnen, da die jeweils hinzu kommenden Operationen schnell die arithmetischen Pipelines auslasten und eine Verbesserung der Ergebnisrate deshalb nicht mehr möglich ist. Aus diesem Grunde ist hier die direkte Lösung mit den normalen Überlappungsmöglichkeiten vorzuziehen.

		1	2	3	4	5	6	7	8	9	10	11	12
DATA MEMORY	MI												X_2
	MA	B	X_0	A	X_1	F_2					F_3		X_2
			B	X_0	A	X_1	F_2					F_3	
				B	X_0	A	X_1	F_2					F_3
	MD				B	X_0	A	X_1	F_2				
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX				0<B			1<X_1					1<X_2
	DPY					1<X_0	0<A						
DATA BUS	DB												
MULTIPLIER	FMUL					B=X_0		A=X_1			B=X_1		A=X_2
							B=X_0		A=X_1			B=X_1	
								B=X_0		A=X_1			B=X_1
	FM								B=X_0		A=X_1		
ADDER	FADD								S_{21}		S_{22}		
										S_{21}		S_{22}	
	FA										S_{21}		X_2
SPAD	SPFN	B	X_0	A	X_1	F_2					F_3		X_2
	BR												

Bild 4.11: Reservierungstafel für die Berechnung von $x_i = ax_{i-1}+bx_{i-2}+f_i$

mit $S_{i1} = f_i+bx_{i-2}$ und $S_{i2} = S_{i1}+ax_{i-1}$

		13	14	15	16	17	18	19	20	21	22
DATA MEMORY	MI					X_3					X_4
	MA			F_4		X_3			F_5		X_4
					F_4					F_5	
						F_4					F_5
	MD	F_3					F_4				
TABLE MEMORY	TMA										
	TM										
DATA PADS	DPX					$1<X_3$					$1<X_4$
	DPY										
DATA BUS	DB										
MULTIPLIER	FMUL			$B=X_2$		$A=X_3$			$B=X_3$		$A=X_4$
		$A=X_2$			$B=X_2$		$A=X_3$			$B=X_3$	
			$A=X_2$			$B=X_2$		$A=X_3$			$B=X_3$
	FM	$B=X_1$		$A=X_2$			$B=X_2$		$A=X_3$		
ADDER	FADD	S_{31}		S_{32}			S_{41}		S_{42}		
			S_{31}		S_{32}			S_{41}		S_{42}	
	FA			S_{31}		X_3			S_{41}		X_4
SPAD	SPFN			F_4		X_3			F_5		X_4
	BR										

Bild 4.11: (Fortsetzung)

		1	2	3	4	5	6	7	8	9	10	11	12
DATA MEMORY	MI												
	MA	A	B	X_0	X_1		F_2			F_3			F_4
			A	B	X_0	X_1		F_2			F_3		
				A	B	X_0	X_1		F_2			F_3	
	MD				A	B	X_0	X_1		F_2			F_3
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX				0<A			-1<X_1	1<K_2	2<F_2			2<F_3
	DPY					0<B	-1<X_0			1<K_1			
DATA BUS	DB												
MULTIPLIER	FMUL				A=A	A=B	B=X_0		A=X_1	A=F_2	K_2=X_0	K_1=X_1	A=F_3
						A=A	A=B	B=X_0		A=X_1	A=F_2	K_2=X_0	K_1=X_1
							A=A	A=B	B=X_0		A=X_1	A=F_2	K_2=X_0
	FM							A=2	A=B	B=X_0		A=X_1	A=F_2
ADDER	FADD							A=2+B		BX_0+F_2		AX_1+BX_0	S_{31}
									A=2+B		BX_0+F_2		AX_1+BX_0
	FA									K_1		BX_0+F_2	
SPAD	SPFN	A	B	X_0	X_1		F_2			F_3			F_4
	BR												

Bild 4.12: Reservierungstafel für die Berechnung von (4.32)

$$x_i = (a^2+b)x_{1-2} + abx_{i-3} + af_{i-1} + f_i$$

mit $K_1 = a^2+b$, $K_2 = ab$, $S_{i1} = af_{i-1}+f_i$, $S_{i2} = S_{i1}+K_2x_{i-3}$

und $S_{i3} = S_{i2} + K_1x_{i-2}$

		13	14	15	16	17	18	19	20	21	22	23	24
DATA MEMORY	MI	X_2					X_3			X_4			X_5
	MA	X_2		F_5			X_3			X_4			X_5
		F_4			F_5								
			F_4			F_5							
	MD			F_4			F_5						
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX	$0<K_2X_0$		$2<F_4$	$0<K_2X_1$		$-1<X_3$	$0<K_2X_2$					
	DPY	$-1<X_2$											
DATA BUS	DB												
MULTIPLIER	FMUL	$K_2=X_1$	$K_1=X_2$	$A=F_4$	$K_2=X_2$	$K_1=X_3$							
		$A=F_3$	$K_2=X_1$	$K_1=X_2$	$A=F_4$	$K_2=X_2$	$K_1=X_3$						
		$K_1=X_1$	$A=F_3$	$K_2=X_1$	$K_1=X_2$	$A=F_4$	$K_2=X_2$	$K_1=X_3$					
	FM	$K_2=X_0$	$K_1=X_1$	$A=F_3$	$K_2=X_1$	$K_1=X_2$	$A=F_4$	$K_2=X_2$	$K_1=X_3$				
ADDER	FADD		S_{32}	S_{41}	S_{33}	S_{42}	S_{51}	S_{43}	S_{52}		S_{53}		
		S_{31}		S_{32}	S_{41}	S_{33}	S_{42}	S_{51}	S_{43}	S_{52}		S_{53}	
	FA	X_2	S_{31}		S_{32}	S_{41}	X_3	S_{42}	S_{51}	X_4	S_{52}		X_5
SPAD	SPFN	X_2		F_5			X_3			X_4			X_5
	BR												

Bild 4.12: (Fortsetzung)

Ebenso liegt der Fall bei Differenzengleichungen mit nicht periodischen Koeffizientenvektoren

$$x_i = a_{i_1} x_{i-1} + a_{i_2} x_{i-2} + \ldots + a_{i_m} x_{i-m} \qquad (4.34)$$

Hier müssen die Koeffizienten der Reduktionsformeln für jedes x wieder neu berechnet werden und sprengen daher schnell den Rahmen der Möglichkeiten, die die vorliegenden 2- und 3-stufigen Pipelines bieten.

Rechner mit einer größeren Stufenzahl ihrer Pipeline-Funktionseinheiten erlauben im allgemeinen mehr Reduktionsschritte und entsprechend mehr Operationen zur Koeffizientenberechnung, bis die Pipelines voll ausgelastet sind, und ermöglichen dadurch schnellere Lösungen der zuletzt angesprochenen Probleme.

4.3.4 Linear rekurrente Systeme

Eine weitere Gruppe rekursiver Probleme wird gebildet durch die linear rekurrenten Systeme.

Definition:
Ein linear rekurrentes System R(n,m) der Ordnung m für n Gleichungen ist definiert für $m \leq n-1$ durch

$$x_k = \begin{cases} 0 & \text{für } k \leq 0 \\ c_k + \sum_{j=k-m}^{k-1} a_{kj} x_j & \text{für } 1 \leq k \leq n \end{cases} \qquad (4.35)$$

Ist $m = n-1$, so spricht man von dem gewöhnlichen linear rekurrenten System R(n).

Zur Berechnung des Lösungsvektors $X = (x_1, \ldots, x_n)$ eines gewöhnlichen linear rekurrenten Systems werden also benötigt:

- ein Konstantenvektor $C = (c_1, \ldots, c_n)$ und
- eine Koeffizientenmatrix $A = (a_{ij})^n$, $1 \leq i,j \leq n$ mit $a_{ij} = 0$ für $i \leq j$ (untere Dreiecksmatrix)

Als Beispiel sei hier das Gleichungssystem R(6) angegeben:

$$x_1 = c_1$$

$$x_2 = c_2 + a_{21} x_1$$

$$x_3 = c_3 + a_{31} x_1 + a_{32} x_2$$

$$x_4 = c_4 + a_{41} x_1 + a_{42} x_2 + a_{43} x_3$$

$$x_5 = c_5 + a_{51} x_1 + a_{52} x_2 + a_{53} x_3 + a_{54} x_4$$

$$x_6 = c_6 + a_{61} x_1 + a_{62} x_2 + a_{63} x_3 + a_{64} x_4 + a_{65} x_5$$

Der "Column-Sweep Algorithmus" /Kuc 78/, entwickelt für die Parallelverarbeitung, erzielt für solche Gleichungssysteme sehr effiziente Lösungen. Dieser Algorithmus basiert auf der Verwendung der Broadcast-Funktion, das heißt, es besteht die Möglichkeit, ein Datum gleichzeitig an mehrere parallel arbeitende Funktionseinheiten oder Prozessoren zu schicken.

Die Arbeitsweise des Algorithmus läßt sich in drei Schritten kurz darstellen:

1. $x_i = c_i$ für $1 \leq i \leq n$; $j := 1$
2. $a_{ij} x_j = p_i$ für $2 \leq i \leq n$, gleichzeitig auf n-1 Prozessoren
3. $x_i = x_i + p_i$ für $2 \leq i \leq n$ gleichzeitig; $j := j + 1$

Das Gleichungssystem wird also spalten- und nicht zeilenweise abgearbeitet. Nach erfolgter Berechnung einer Spalte (Multiplikation + Addition zum bereits berechneten Wert) steht jeweils ein neuer x-Wert zur Verfügung, der dann zur Rechnung in der nächsten Spalte benötigt wird. So wird hier sehr einfach die Rekursion umgangen, denn bei zeilenweiser Rechnung muß jeweils die Vorgängerzeile vollständig berechnet sein, bevor die nächste Zeile abgearbeitet werden kann.

Ähnlich diesem Prinzip läßt sich auch eine effiziente Lösungsmöglichkeit für Pipelinerechner angeben. Hier wird ebenfalls spaltenweise gearbeitet, nur mit dem Unterschied, daß die Berechnungen einer Spalte hintereinander - pro Takt eine neue Berechnung - in die arithmetische Pipeline geschoben werden.

Für die Lösung auf dem AP ergibt sich dann die Vorgehensweise nach der Reservierungstafel in Bild 4.13. Zur allgemeinen Lösung gewöhnlicherlinear rekurrenter Systeme R(N) mit $3 \leq N \leq 32$ sei auf das Assemblerprogramm im Anhang hingewiesen.

		1	2	3	4	5	6	7	8	9	10	11	12
DATA MEMORY	MI												
	MA	C_1	C_2	C_3	C_4	C_5	C_6	A_{21}	A_{31}	A_{41}	A_{51}	A_{61}	A_{32}
			C_1	C_2	C_3	C_4	C_5	C_6	A_{21}	A_{31}	A_{41}	A_{51}	A_{61}
				C_1	C_2	C_3	C_4	C_5	C_6	A_{21}	A_{31}	A_{41}	A_{51}
	MD				C_1	C_2	C_3	C_4	C_5	C_6	A_{21}	A_{31}	A_{41}
TABLE MEMORY	TMA												
	TM												
DATA PADS	DPX				$-2<C_1$	$-1<C_2$	$0<C_3$	$1<C_4$	$2<C_5$	$3<C_6$			
	DPY				$-2<X_1$								
DATA BUS	DB												
MULTIPLIER	FMUL										$A_{21}*X_1$	$A_{31}*X_1$	$A_{41}*X_1$
												$A_{21}*X_1$	$A_{31}*X_1$
													$A_{21}*X_1$
	FM												
ADDER	FADD												
	FA												
SPAD	SPFN												
	BR												

Bild 4.13: Reservierungstafel zur Berechnung von R(6)

mit $S_{ij} = a_{ij}x_j = C_{ij}$

		13	14	15	16	17	18	19	20	21	22	23	24
DATA MEMORY	MI												
	MA	A_{42}	A_{52}	A_{62}		A_{43}	A_{53}	A_{63}			A_{54}	A_{64}	
		A_{32}	A_{42}	A_{52}	A_{62}		A_{43}	A_{53}	A_{63}			A_{54}	A_{64}
		A_{61}	A_{32}	A_{42}	A_{52}	A_{62}		A_{43}	A_{53}	A_{63}			A_{54}
	MD	A_{51}	A_{61}	A_{32}	A_{42}	A_{52}	A_{62}		A_{43}	A_{53}	A_{63}		
TABLE MEMORY	TMA												
	TM												
DATA PROS	DPX				$0<C_{31}$	$1<C_{41}$	$2<C_{51}$	$3<C_{61}$		$1<C_{42}$	$2<C_{52}$	$3<C_{62}$	
	DPY			$-1<X_1$					$0<X_3$				
DATA BUS	DB												
MULTIPLIER	FMUL	$A_{51}=X_1$	$A_{61}=X_1$	$A_{32}=X_2$	$A_{42}=X_2$	$A_{52}=X_2$	$A_{62}=X_2$		$A_{43}=X_3$	$A_{53}=X_3$	$A_{63}=X_3$		
		$A_{41}=X_1$	$A_{51}=X_1$	$A_{61}=X_1$	$A_{32}=X_2$	$A_{42}=X_2$	$A_{52}=X_2$	$A_{62}=X_2$		$A_{43}=X_3$	$A_{53}=X_3$	$A_{63}=X_3$	
		$A_{31}=X_1$	$A_{41}=X_1$	$A_{51}=X_1$	$A_{61}=X_1$	$A_{32}=X_2$	$A_{42}=X_2$	$A_{52}=X_2$	$A_{62}=X_2$		$A_{43}=X_3$	$A_{53}=X_3$	$A_{63}=X_3$
	FM	$A_{21}=X_1$	$A_{31}=X_1$	$A_{41}=X_1$	$A_{51}=X_1$	$A_{61}=X_1$	$A_{32}=X_2$	$A_{42}=X_2$	$A_{52}=X_2$	$A_{62}=X_2$		$A_{43}=X_3$	$A_{53}=X_3$
ADDER	FADD	S_{21}	S_{31}	S_{41}	S_{51}	S_{61}	S_{32}	S_{42}	S_{52}	S_{62}		S_{43}	S_{53}
			S_{21}	S_{31}	S_{41}	S_{51}	S_{61}	S_{32}	S_{42}	S_{52}	S_{62}		S_{43}
	FA			X_2	C_{31}	C_{41}	C_{51}	C_{61}	X_3	C_{42}	C_{52}	C_{62}	
SPAD	SPFN												
	BR												

Bild 4.13: (Fortsetzung 1)

		25	26	27	28	29	30	31	32	33	34	35	
DATA MEMORY	MI												
	MA			A_{65}									
					A_{65}								
		A_{64}				A_{65}							
	MD	A_{54}	A_{64}				A_{65}						
TABLE MEMORY	TMA												
	TM												
DATA PROS	DPX		$2<C_{53}$	$3<C_{63}$				$3<C_{64}$					
	DPY	$1<X_4$					$2<X_5$						
DATA BUS	DB												
MULTIPLIER	FMUL	$A_{54}*X_4$	$A_{64}*X_4$				$A_{65}*X_5$						
			$A_{54}*X_4$	$A_{64}*X_4$				$A_{65}*X_5$					
				$A_{54}*X_4$	$A_{64}*X_4$				$A_{65}*X_5$				
	FM	$A_{63}*X_3$			$A_{54}*X_4$	$A_{64}*X_4$				$A_{65}*X_5$			
ADDER	FADD	S_{63}			S_{54}	S_{64}				S_{65}			
		S_{53}	S_{63}			S_{54}	S_{64}				S_{65}		
	FA	X_4	C_{53}	C_{63}			X_5	C_{64}				X_6	
SPAD	SPFN												
	BR												

Bild 4.13: (Fortsetzung 2)

Hier sind also nicht, wie bei den Differenzengleichungen in Abschnitt 4.3.3 aufwendige Problemumformungen erforderlich, um eine effiziente Ausnutzung der vorliegenden Pipelinestruktur zu erreichen, sondern es genügt eine Änderung der Reihenfolge in der Ausführung der gewünschten arithmetischen Operationen.

Von den in diesem Kapitel aufgezeigten speziellen rekursiven Problemen läßt sich leicht ein Übergang finden zu den rechenintensiven Problemen, die sich im Bereich der Numerischen Mathematik ergeben. Auch hier werden meist durch Rekursionen bestimmte Folgen- und Näherungswerte berechnet.

4.4 Höhere Programmiersprachen für Pipelinerechner

Die Beispiele der vorangegangenen Abschnitte wurden unter Verwendung der Assemblersprache des AP gelöst, denn so bestand die Möglichkeit, genaue Angaben über das Zeitverhalten der Maschine zu machen. Für den Anwendungsprogrammierer ist diese Methode jedoch normalerweise nicht sinnvoll, da sie eine genaue Kenntnis der Hardware voraussetzt. Hier erfolgt also die Kommunikation mit dem Rechner über die höheren Programmiersprachen.

Bei konventionellen höheren Programmiersprachen erfolgt die Verarbeitung von Vektoren und Matrizen mit Hilfe von DO-Loops, die zunächst keine effiziente Nutzung der Pipeline- Organisation erlauben. So müssen im Prinzip für jedes Operandenpaar folgende Operationen vollständig ausgeführt sein, bevor ein neues Operandenpaar bearbeitet werden kann:

1. Holen des 1. Operanden
2. Holen des 2. Operanden
3. Ausführen der geforderten Operation
4. Abspeichern der Ergebnisse
5. Erhöhen des Index
6. Abfrage und Verzweigung.

Ein Pipelinerechner hat aber die Möglichkeit, durch Überlappung der einzelnen Operationen die Berechnung zu beschleunigen, so daß maximal pro Takt ein Ergebnis berechnet wird. Um diese Vorteile der Pipeline Verarbeitung auch beim Gebrauch einer höheren Programmiersprache nutzen zu können, sind drei unterschiedliche Vorgehensweisen anwendbar:

- Entwicklung neuer Programmiersprachen
- Erweiterung bestehender Programmiersprachen
- Verbesserungen der Compiler.

Die Entwicklung neuer Programmiersprachen für Pipeline- und Parallelrechner zielt in erster Linie darauf hin, Vektoren und Vektoroperationen explizit dem Rechner kenntlich zu machen. Dazu hat der Programmierer die Möglichkeit, Datenstrukturen als Vektoren oder Felder zu identifizieren und die Verarbeitung aller Elemente durch eine einzige Instruktion ausführen zu lassen. Hierbei ist zu beachten, daß die entsprechenden Elemente in aufeinanderfolgenden Speicherplätzen untergebracht sind, um so einen reibungslosen Datenfluß, ohne aufwendige Adreßrechnung, gewährleisten zu können. Beispiele solcher Sprachen zur Vektorverarbeitung sind APL und VECTRAN /PaW 75/.

Ähnlich wie im Fall der Neuentwicklung von Programmiersprachen liegt auch die Zielsetzung bei der Erweiterung bestehender Sprachen. Bei der CYBER 205 besteht so zum Beispiel durch eine Erweiterung des Standard-FORTRAN die Möglichkeit, Vektoren und Vektoroperationen explizit anzugeben /Cyb 82/. Es genügt dann, dem Vektorprozessor jeweils die Anfangsadresse und die Länge der Vektoren mitzuteilen, um mit der überlappten Bearbeitung der einzelnen Operationen aus dem obigen Beispiel beginnen zu können.

Durch die hier angesprochenen Neuerungen ist auch bei der Programmierung in einer höheren Programmiersprache die Ausnutzung einer Pipeline-Organisation erreichbar. Ein Problem bildet allerdings die Fülle der bereits existierenden Programme, die für konventionelle Maschinen entwickelt wurden. Diese können bei weitem nicht alle unter Verwendung neuer Sprachkonstrukte umprogrammiert werden. Um aber auch für konventionell geschriebene Programme effiziente Lösungen zu erzielen, wurden die sogenannten "vektorisierenden" Compiler entwickelt.

Vektorisieren heißt, der Compiler untersucht die Schleifen eines konventionell geschriebenen Programms und formt sie gegenenenfalls in Vektoroperationen um. Die FORTRAN Compiler sowohl der CYBER 205 als auch der CRAY-1 führen automatisch solche Vektorisierungen durch.

Durch Compilerdirektiven können dem FORTRAN-Compiler der CRAY-1 spezielle Informationen für diese Vektorisierungen mitgeteilt werden /CRA 81/. Dies erfolgt durch spezielle Kommentarzeilen, die in das normale Programm eingefügt werden. Daher bleiben die Programme auch auf anderen Rechnern ausführbar. Die Vektor-Direktive zum Beispiel veranlaßt den Compiler, innere Schleifen mit mehr als 5 Durchläufen zu vektorisieren. Aufgrund der längeren Vorbereitungszeit für Vektorregister als für Skalarregister werden Schleifen mit geringerer Durchlaufzahl schneller im Skalar-Modus abgearbeitet. Weitere Direktiven erlauben auch Informationen über Schleifen, welche Rekursionen beinhalten.

Allerdings ist es derzeit auch diesem sehr komplexen CRAY-Compiler nur möglich, solche Vektorisierungen in beschränktem Maße durchzuführen, das heißt, nur die innersten Schleifen könne vektorisiert werden, und auch nur dann, wenn sie in Zusammenhang mit Array-Operationen auftreten, also weder Verzweigungen, noch Bedingungen, noch Unterprogrammaufrufe enthalten.

5. Schlußbetrachtungen

Ausgehend von der Forderung nach hohen Rechenleistungen, in erster Linie aus den Bereichen der naturwissenschaftlich-technischen Forschung (number crunching) und der Realzeit Verarbeitung großer Datenmengen, wurden, unter der Verwendung neuer Technologien, wesentliche Geschwindigkeits-steigerungen durch innovative Architekturkonzepte erzielt, die teilweise erheblich von der konventionellen von-Neumann Struktur abweichen.

Das Prinzip des Pipelining stellt sich hier als wichtige Architekturform moderner Hochleistungsrechner dar. Wie die Zusammenstellung in Kapitel 2 zeigt, ist nahezu die Hälfte aller neuen Architekturen von der Idee der Pipeline-verarbeitung beeinflußt.

Der Hardware-Aufbau solcher Pipeline-Organisationen und die Kontrolle des Datenflusses durch eine Pipeline bilden die Schwerpunkte in Kapitel 3. Reservierungstafeln erweisen sich hier als gute Hilfsmittel, um den Weg der Daten durch eine Pipeline zu verfolgen und eventuelle Konflikte festzustellen.

Als Möglichkeit der Kontrolle auch komplexer Pipeline-Organisationen lassen sich daher die aus den Reservierungstafeln resultierenden Kollisionsvektoren und Zustandsdiagramme für statische und dynamische Pipelines verwenden. Dies konnte hier nur theoretisch erörtert werden, und in der Praxis muß sich zeigen, ob diese Art der Kontrolle auch effizient eingesetzt werden kann.

Das Kapitel "Programmierung" befaßt sich in erster Linie mit rekursiven Problemen, die aufgrund ihrer Struktur zunächst einmal der Idee des Pipelining widersprechen. Durch spezielle Problemumformungen (Rekursionsreduktionen) läßt sich jedoch auch bei diesen Problemen unter Ausnutzung der vorliegenden Pipelinestruktur eine Steigerung der Rechenleistung erzielen. Diese Resultate kamen vorwiegend durch Probieren zustande, und es stellt sich die Frage, ob in Zukunft eine gewisse Automatisierung dieser Problemumformungen erreicht werden kann.

Für die Programmierung in einer höheren Programmiersprache sind ebenfalls die Probleme bei weitem noch nicht als gelöst zu betrachten. Die Entwicklung geeigneter Sprachkonstrukte sowie neuer Compilertechniken wird auch weiterhin ein Hauptproblem für die effiziente Programmierung von Pipelinerechnern bleiben.

6. Literaturverzeichnis

/Bac 78/ Backus J.
"Can Programming be Liberated from the von-Neumann Style? A Functional Style and Its Algebra of Programms"
Comm. ACM pp. 613 - 641, August 1978

/BBK 68/ Barnes G. H., Brown R. M., Kuck D.
"The ILLIAC IV Computer"
IEEE Trans. on Computers, Vol C-17, No. 8, pp. 746 - 757, August 1968

/Ber 80/ Berg L.
"Differenzengleichungen 2. Ordnung mit Anwendungen"
UTB 906, 1980

/BoH 80/ Bode A., Händler W.
"Rechnerarchitektur"
Springer Verlag 1980

/Che 71/ Chen T. C.
"Parallelism, Pipelining and Computer Efficiency"
Computer Design, pp. 69 - 74, January 1971

/CRA 81/ "CRAY-1 FORTRAN Reference Manual"
CRAY Research Inc. 1981

/Cyb 82/ "Die CDC CYBER 205"
Bericht des Rechenzentrums der RWTH Aachen 1982

/Dav 71/ Davidson E. S.
"The Design and Control of Pipelined Function Generators"
Proc. Int. IEEE Conf. on Systems, Networks and Computers, Oaxtepec, Mexico, January 1971

/Dor 79/ Doran R. W.
"Computer Architekture: A Structured Approach"
Academic Press 1979

/Fly 72/ Flynn M. J.
"Some Computer Organizations and Their Effectiveness"
IEEE Trans. on Computers, Vol. C-21, pp. 948 - 960, Sept. 1972

/FPS 78/ "Programmers Reference Manual"
Floating Point Systems Inc. 1978

/Gil 81/ Giloi W. K.
"Rechnerarchitektur"
Springer Verlag 1981

/Hae 77/ Händler W.
"On Classification Schemes for Computersystems in the Post von Neumann Era"
Lecture Notes in Comp. Science 26, Springer Verlag 1975, pp.439 - 452

/HaT 72/ Hintz R. G., Tate D. P.
"Control Data STAR 100 Processor Design"
COMPCON 1972, IEEE Publication 1972, 1 - 4

/Hoß 80/ Hoßfeld F.
"Parallelprozessoren und Algorithmenstruktur"
Spezielle Berichte der KFA Jülich Nr.87, September 1980

/IBM 76/ "IBM 3838 AP Functional Characteristics"
IBM Corp. No.6A24-3639-0, Endicott, N.Y. October 1976

/Kla 75/ Klar R.
"Digitale Rechenautomaten"
Sammlung Göschen, de Gruyter 1976

/Kog 73/ Kogge P. M.
"Maximal Rate Pipelined Solutions To Recurrence Problems"
Proc. 1 st Annual Conf. on Computer Architektur 1973

/Kog 81/ Kogge P. M.
"The Architecture of Pipelined Computers"
Mc Graw Hill 1981

/Kuc 78/ Kuck D. J.
"The Structure of Computers and Computations"
John Wiley & Sons 1978

/Kuc 80/ Kuck D. J.
"High Speed Machines and Their Compilers"
CREST Conference September 1980

/Mye 78/ Myers G. J.
"Advances in Computer Architecture"
John Wiley & Sons 1978

/PaW 75/ Paul G., Wilson M. W.
"The VECTRAN Language: An Experimental Languagefor Vector/Matrix Array Processing"
IBM Palo Alto Scientific Center Report 6320-3334, August 1975

/RaL 77/ Ramamoorthy C. V., Li H. F.
"Pipeline Architecture"
ACM Computing Surveys Vol.9 No.1 March 1977
pp. 61 - 102

/ReF 76/ Reddi, Feustel
"A Conceptual Framework for Computer Architecture"
Computing Surveys Vol.8 No.2 June 1976

/Rus 78/ Russel R. M.
"The CRAY-1 Computer System"
Comm. ACM January 1978 pp. 63 - 72

/SFS 77/ Swan R. J., Fuller S. H., Siewiorek D. P.
"Cm* - A Modular, Multi-Microprocessor"
Proc. AFIPS NCC 1979

/Sha 72/ Shar L. E.
"Design and Sceduling of Statically Configured Pipelines"
Digital Systems, Lab. Report SU-SEL-72-042, Stanford University California, September 1972

/Smi 79/ Smith B. J.
"A Pipelined, Shared Resource MIMD Computer"
Proc. of the 1978 Int'1. Conf. on Parallel Processing, Aug. 1978

/Thu 76/ Thurber K. J.
"Large Scale Computer Architecture"
Hayden Book Company 1976

/WaB 72/ Wulf W. A., Bell C. G.
"C.mmp - A Multi-Mini-Processor"
Proc. AFIPS FJCC 1972 pp. 765 - 777

A n h a n g

Kurzbeschreibung der benötigten APAL-Befehle

MOV i,j	Der Inhalt des S-Pad Registers i wird noch im gleichen Takt auf SPFN gelegt und im S-Pad Register j abgespeichert
INC i	Der Inhalt des S-Pad Registers i wird um 1 erhöht
DEC i	Der Inhalt des S-Pad Registers i wird um 1 verringert
SETMA	Setzen des MA Registers von SPFN und Starten eine Speicherzyklus
INCMA	Der Inhalt des MA Registers wird um 1 erhöht und ein Speicherzyklus gestartet
DECMA	Der Inhalt des MA Registers wird um 1 verringert und ein Speicherzyklus gestartet
DPX(i)<j	Das i-te DPX-Register, relativ zum Pointer, wird mit dem Inhalt des Registers j geladen
DPY(i)<j	Analog zu DPX
FMUL i,j	Starten einer Multiplikation der Inhalte der beiden Register i und j sowie Weiterschieben bereits begonnener Berechnungen
FADD i,j	Starten einer Addition der Inhalte der beiden Register i und j sowie Weiterschieben bereits begonnener Berechnungen durch die Pipeline
SETDPA	Setzen des DPA Registers (Pointer der Data Pads) von SPFN
INCDPA	Erhöhen des Inhaltes von DPA um 1
DECDPA	Vermindern des Inhaltes von DPA um 1
LDDA	Laden des Device Address Register vom DAta Pad Bus (wird benötigt um in den Tabellenspeicher schreiben zu können)
SETTMA	Setzen des TMA Registers und Starten eines Tabellenspeicher-Zyklus
JMP l	Sprung zur symbolischen Adresse l
BGT l	Sprung zur symbolischen Adresse l, falls SPFN im vorhergehenden Takt größer als Null war
BGE l	Sprung zur symbolischen Adresse l, falls SPFN im vorhergehenden Takt größer oder gleich Null war

BR 1 Sprung zur symbolischen Adresse 1

NOP Keine Instruktionsausführung in diesem Takt

In den folgenden Programmlisten sind alle Befehle, die in einem Takt ausgeführt werden, durch Semikolon voneiander getrennt.

Die Belegung der S_PAD Register muß jeweils vor Beginn der Programmausführung über den Host-Rechner erfolgen.

```
" APAL-UNTERPROGRAMM ZUR BERECHNUNG HOMOGENER DIFFERENZENGLEICHUNGEN
"          2.ORDNUNG MIT KONSTANTEN KOEFFIZIENTEN
"
      $TITLE DIFF
      $ENTRY DIFF,5
"
"     BELEGUNG DER S-PAD REGISTER
"
      ANZ   $EQU 0     " ANZAHL DER ZU BERECHNENDEN FOLGENELEMENTE
      A     $EQU 1     " ADRESSE DES KOEFFIZIENTEN A
      B     $EQU 2     " ADRESSE DES KOEFFIZIENTEN B
      X0    $EQU 3     " ADRESSE DES ANFANGSWERTES X0
      X1    $EQU 4     " ADRESSE DES ANFANGSWERTES X1
"
DIFF: MOV A,A; SETMA
      MOV B,B; SETMA
      MOV X1,X1;SETMA
      DPX(0) < MD ;MOV X0,X0;SETMA                "DPX(0)=A
      DPY(0) < MD                                 "DPY(0)=B
      DPY(1) < MD ;FMUL DPX(0),DPX(0)             "DPY(1)=X1
      DPX(1) < MD ;FMUL DPX(0),DPY(1)             "DPX(1)=X0
      FMUL DPX(1),DPY(0); FADD DPY(0),DPY(0)
      FMUL DPX(0),DPY(0); FADD FM,DPY(0);         "DPX(2)=A**2
      DPX(2) < FM
      FMUL; FADD DPX(2),FA; DPX(-1) < FM          "DPX(-1)=A*X1
      FMUL DPY(1),FA; FADD FM,DPX(-1);            "DPX(-2)=A**2+B
                      DPX(-2) < FA
      FMUL FM,DPX(1); FADD; DPY(2) < FA           "DPY(2)=A**2+2B
      FMUL DPY(2),DPX(0); DPY(2) < FA             "DPY(2)=X2
      FMUL DPY(0),DPX(-2); DPX(0) < FM            "DPX(0)=(A**2+B)X1
      FMUL; FADD FM,DPX(0)
      FMUL FM,DPY(1); DPX(0) < FM;
                    INC X1; SETMA; MI < FM        "DPX(0)=A**3+2AB
      FMUL FM,DPX(1);  DPX(2) < FM;               "DPX(2)=A**2*B+B**2
                   INCMA; MI< FM
      FMUL DPX(0),DPY(2)
      FMUL DPX(2),DPY(1); DPY(-1) < FM;           "DPY(-1)=K1*X1
                      FADD
LOOP: FMUL DPX(0),FA; FADD FM,DPY(-1);            "DPY(0)=X(2N+1)
           DPY(0) < FA;INCMA; MI<FA
      FMUL DPX(2),DPY(2); DPY(-1) < FM; FADD      "DPY(-1)=K2X(2N)
      FMUL DPX(0),FA; FADD FM,DPY(-1);
           DPY(2) < FA; DEC ANZ; INCMA;MI < FA    "DPY(2)=X(2(N+1))
      FMUL DPX(2),DPY(0); DPY(-1) < FM;           "DPY(-1)=K1X(2N+1)
                      FADD; BNE LOOP
      RETURN
      $END
```

Zur folgenden Berechnung gewöhnlicher linearer rekurrenter Systeme (GLRS) zunächst noch einige Bemerkungen.

Das APAL-Programm berechnet GLRS R(N) mit $N < 33$. Dies resultiert daher, daß die 32 Register des DPY Blockes zur Zwischenspeicherung der Teilergebnisse c_{ij} sowie der Ergebnisse x_i benötigt werden. Das Zwischenspeichern auch der Endresultate wird erforderlich, da pro Takt nur ein Hauptspeicherzugriff (Lesen oder Schreiben) erfolgen kann, der schon durch das Holen der Koeffizienten a_{ij} ausgeschöpft wird.

Für $N \geq 33$ sieht die Lösung ähnlich aus, nur muß jetzt nach der Berechnung von jeweils 32 Teilergebnissen ein Umspeichern dieser Daten erfolgen.

Die Berechnung beginnt mit dem Laden des C-Vektors in die DPY Register. Vor- und Nachlauf der dies erledigenden LOOP-Schleife sind erforderlich, um sicherzustellen, daß jeweils nicht mehr als N Werte ausgelesen werden.

Bei den Berechnungen einer neuen Spalte wird unterschieden, ob 1,2,..,8 oder mehr Zeilen zu durchlaufen sind, damit auch hier wieder nur die geforderten Koeffizienten aus dem Hauptspeicher geholt werden. Die Zahl 8 resultiert aus einer Art Makropipeline, die sich aus dem Speicherzyklus (3 Takte), dem Multiplizierer (3 Takte) und dem Addierer (2 Takte) zusammensetzt.

Das TM-Register des Tabellenspeichers übernimmt die Aufgabe, den in der vorhergehenden LOOP1-Schleife berechneten x-Wert für die Berechnungen der nächsten Spalte bereitzustellen. Dies kann nicht durch ein DPX-Register erfolgen, da im DPY Block, wie oben gezeigt, bis zu 32 Werte abgelegt werden müssen, und so der nur relativ zum Pointer mögliche Zugriff nicht mehr ausreicht, um pro Takt, also ohne Umsetzen des Pointers, auf ein spezielles DPX-Register zugreifen zu können.

Nach Beendigung der Berechnungen werden mit der LOOP2-Schleife die Resultate in den Hauptspeicher geladen.

```
 10 "
 20 " APAL UNTERPROGRAMM ZUR BERECHNUNG GEWOEHNLICHER LINEARER
 30 "          REKURRENTER SYSTEME R(N) MIT 3 < N < 33
 40 "
 50         $TITLE GLRS
 60         $ENTRY GLRS,5
 70 "
 80 "       BELEGUNG DER S-PAD REGISTER
 90 "
100           N   $EQU  0          "ANZAHL DER GLEICHUNGEN
110         ADC   $EQU  1          "ANFANGSADRESSE C-VEKTOR
120         ADA   $EQU  2          "ANFANGSADRESSE A-VEKTOR
130         ADX   $EQU  3          "ANFANGSADRESSE X-VEKTOR
140         TAD   $EQU  4          "TABELLENSPEICHERADRESSE
150         ANZ1  $EQU  5          "HILFSZAEHLER
160         ANZ   $EQU  6          "HILFSZAEHLER
170 "
180 "       LADEN DES C-VEKTORS IN DPY
190 "
200  GLRS:MOV N,ANZ
210         DEC ANZ
220         DEC ANZ
230         MOV ADC,ADC; SETMA;SETDPA
240         DEC ANZ ; INCMA
250         DEC ANZ; INCMA
260  LOOP:DPY(0) < MD; DEC ANZ; INCMA; INCDPA; BGT LOOP
270         DPY(0) < MD; INCDPA
280         DPY(0) < MD; INCDPA
290         DPY(0) < MD ; MOV N,ANZ
300 "
310 "       STARTEN DER BERECHNUNG EINER NEUEN SPALTE
320 "
330 LOOP1:MOV ADC,ADC; SETDPA
340         DPX(0) < DPY(0)
350 "
360 "       ZULETZT BERECHNETEN X-WERT IN TABLE MEMORY LADEN
370 "
380         LDDA; DB = 5
390         MOV TAD,TAD; SETTMA; DB = DPX(0); OUT
400 "
410         DEC ANZ
420         MOV ANZ,ANZ1
430         MOV TAD,TAD; SETTMA
440         DEC ANZ1
450         DEC ANZ1
460         MOV ADA,ADA ; SETMA; BGE L1
470 "
480         NOP
490         NOP
500         FMUL TM,MD
510         FMUL
520         FMUL
530         FADD FM,DPY(1)
540         FADD
550         DPY(1) < FA; JMP END
```

```
560 "
570  L1:  ADD ANZ,ADA
580       MOV ANZ1,ANZ1
590       INCMA; DEC ANZ1; BGT L2
600 "
610       NOP
620       FMUL TM,MD
630       FMUL TM,MD
640       FMUL
650       FMUL; FADD FM,DPY(1); INCDPA
660       FADD FM,DPY(1); INCDPA
670       JMP G1
680 "
690    L2: INCMA; DEC ANZ1; BGT L3
700 "
710       FMUL TM,MD
720       FMUL TM,MD
730       FMUL TM,MD
740       FMUL; FADD FM,DPY(1); INCDPA
750       FMUL; FADD FM,DPY(1);INCDPA
760       JMP G2
770 "
780  L3:  FMUL TM,MD; INCMA; DEC ANZ1; BGT L4
790 "
800       FMUL TM,MD
810       FMUL TM,MD
820       FMUL TM,MD; FADD FM,DPY(1); INCDPA
830       FMUL; FADD FM,DPY(1); INCDPA
840       JMP G3
850 "
860    L4: FMUL TM,MD; INCMA; DEC ANZ1; BGT L5
870 "
880       FMUL TM,MD
890       FMUL TM,MD;FADD FM,DPY(1); INCDPA
900       FMUL TM,MD; FADD FM,DPY(1); INCDPA; BR G4
910 "
920  L5:  FMUL TM,MD; INCMA; DEC ANZ1; BGT L6
930 "
940       FMUL TM,MD; FADD FM,DPY(1); INCDPA
950       FMUL TM,MD; FADD FM,DPY(1); INCDPA; BR G5
960 "
970    L6: FMUL TM,MD; INCMA; DEC ANZ1;
980              FADD FM,DPY(1); INCDPA; BGT L7
990 "
1000      FMUL TM,MD; FADD FM,DPY(1); INCDPA; BR G6
1010 "
1020 L7:  FMUL TM,MD; INCMA; DEC ANZ1;
1030             FADD FM,DPY(1); INCDPA; BGT L8
1040 "
1050      JMP G7
```

```
1060 "
1070  L8:  FMUL TM,MD; INCMA; DEC ANZ1;
1080               FADD FM,DPY(1); INCDPA; DPY(-1) < FA; BGT L8
1090
1100  G7:  FMUL TM,MD; FADD FM,DPY(1); INCDPA; DPY(-1) < FA
1110  G6:  FMUL TM,MD; FADD FM,DPY(1); INCDPA; DPY(-1) < FA
1120  G5:  FMUL TM,MD; FADD FM,DPY(1); INCDPA; DPY(-1) < FA
1130  G4:  FMUL; FADD FM,DPY(1); INCDPA; DPY(-1) < FA
1140  G3:  FMUL; FADD FM,DPY(1); INCDPA; DPY(-1) < FA
1150  G2:  FADD FM,DPY(1); INCDPA; DPY(-1) < FA
1160  G1:  FADD; INCDPA; DPY(-1) < FA; INC ADC
1170       DPY(-1) < FA; JMP LOOP1
1180 "
1190 "
1200  END: SUB ADC,ADC; SETDPA
1210       DEC N
1220       DEC N
1230       MOV ADX,ADX; SETMA; MI < DPY(0); INCDPA
1240 LOOP2:INCMA; MI < DPY(0); DEC N; BGT LOOP2; INCDPA
1250       RETURN
1260       $END
```